C MAGAZINE

シューティングゲームアルゴリズムマニアックス

松浦健一郎 著

SOFT BANK Publishing

C MAGAZINE

シューティングゲームアルゴリズムマニアックス

松浦健一郎 著

SOFT BANK
Publishing

シューティングゲーム アルゴリズム マニアックス

SHOOTINGGAME ALGORITHM MANIAX

付録CD-ROMの内容

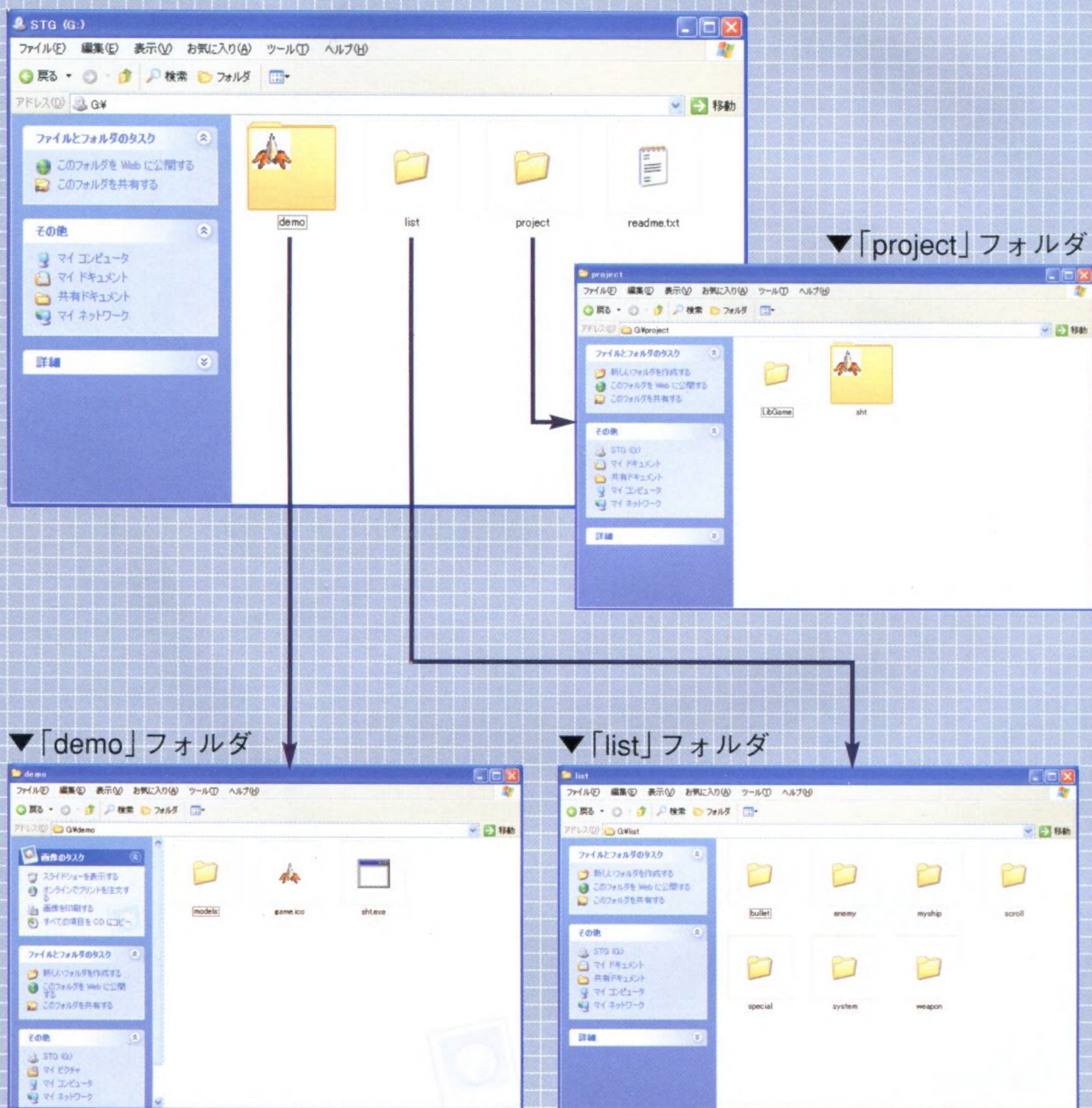
本書の付録CD-ROMには、以下のようなファイルが収録されています。

◇「demo」フォルダ　：実行形式のデモプログラム
◇「list」フォルダ　　：各アルゴリズムを実装したソースコード
◇「project」フォルダ：デモプログラムのプロジェクト一式
◇ readme.txt

各ファイルは、CD-ROMからハードディスク上にコピーして使用してください。
なお、コピーの際には「読み取り専用」属性を解除してください。

▼「project」フォルダ

▼「demo」フォルダ

▼「list」フォルダ

デモプログラムを実行する

付録CD-ROMの「demo」フォルダには、本書内で紹介する各アルゴリズムに対応したデモプログラムが収録されています。「sht.exe」をエクスプローラなどから実行すると、デモプログラムが起動します。各アルゴリズムに対応したデモをカーソルキーで選択することができます。

画面上に表示される操作方法に従ってデモプログラムを動かしてみてください。弾よけや敵の破壊などが楽しめます。

デモプログラムの実行にはDirectX 9.0以上と、対応ビデオカードおよびドライバが必要です。動作確認はWindowsXP/2000/98で行われています。

シューティングゲームアルゴリズムマニアックス デモプログラム Ver.1.00

Stage#1
狙い撃ち弾
自機に向かって飛ぶ弾。

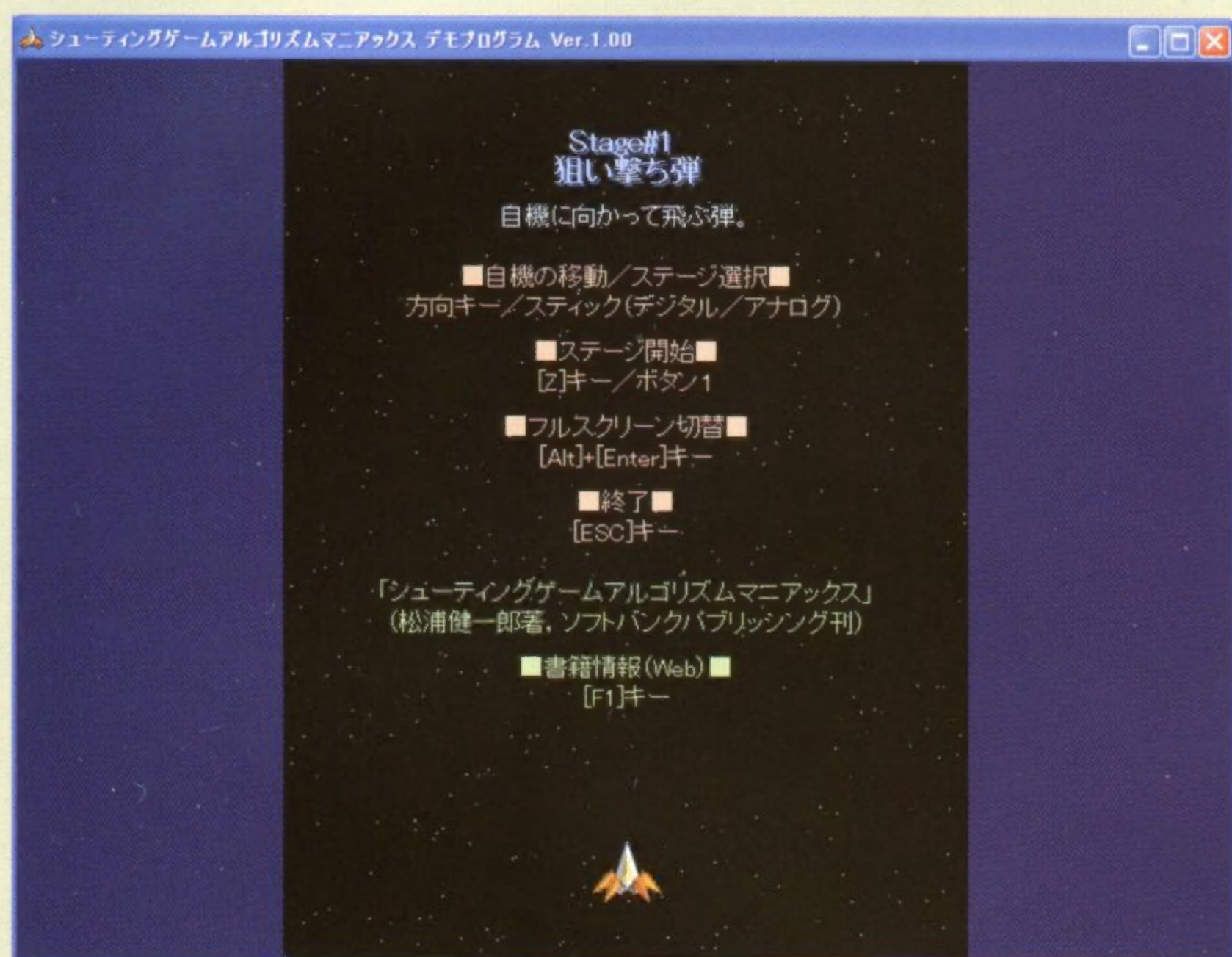
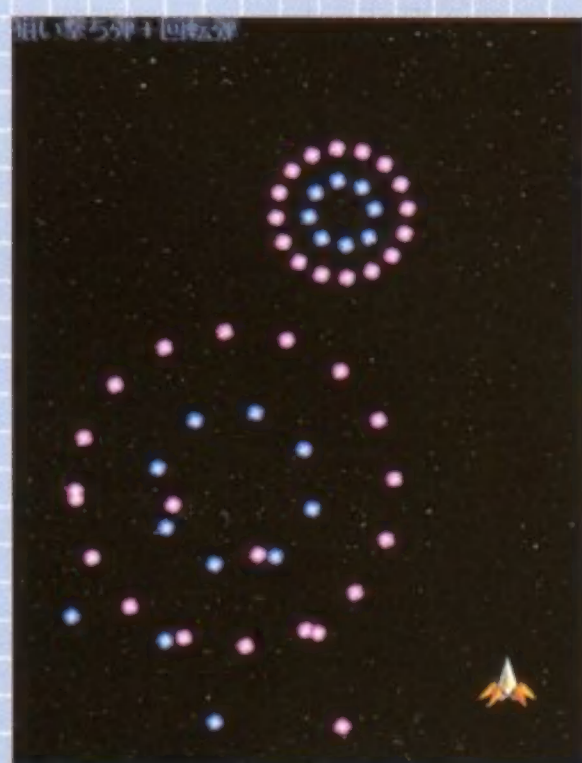
■自機の移動／ステージ選択■
方向キー／スティック(デジタル／アナログ)

■ステージ開始■
[Z]キー／ボタン1

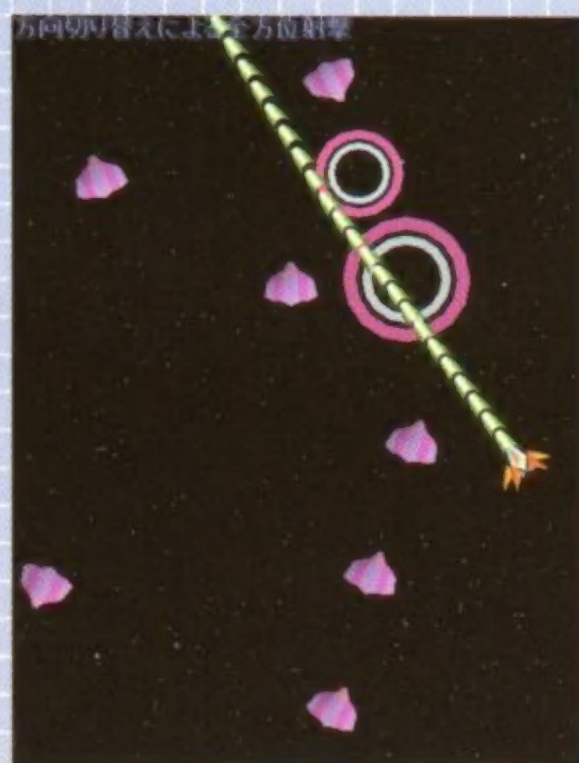
■フルスクリーン切替■
[Alt]+[Enter]キー

■終了■
[ESC]キー

「シューティングゲームアルゴリズムマニアックス」
(松浦健一郎著、ソフトバンクパブリッシング刊)

■書籍情報(Web)■
[F1]キー

◀デモプログラム起動画面

デモプログラムの例

Stage#22
狙い撃ち弾+回転弾

Stage#53
方向切り替えによる
全方位射撃

Stage#77
固定砲台

Stage#80
ボス

デモプログラムのプロジェクト

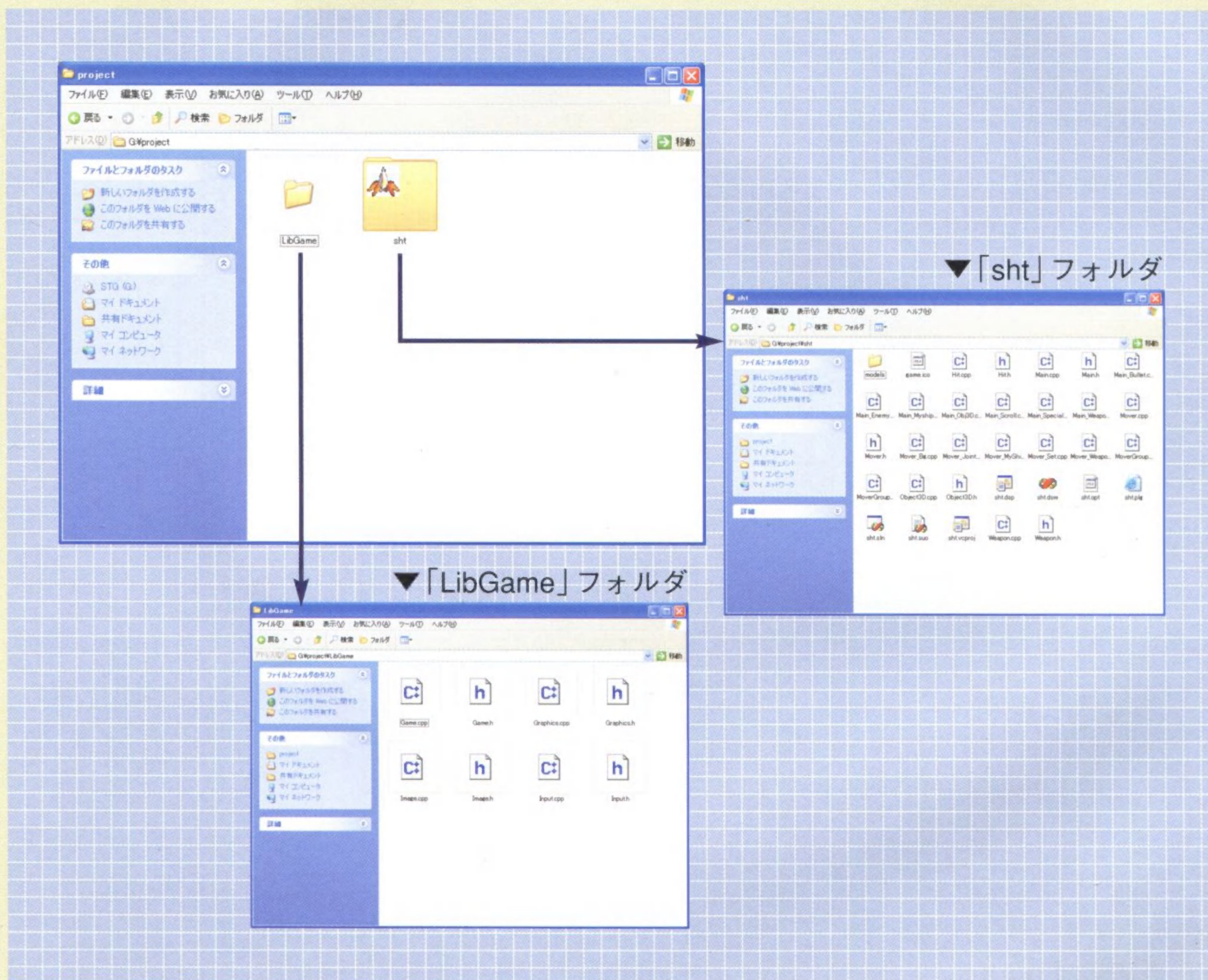
付録CD-ROMの「project」フォルダには、デモプログラムのプロジェクトファイル一式が収録されています。

Visual C++ 6.0用（sht.dsw）とVisual C++ .NET 2003用（sht.sln）のファイルが収録されています。ビルドにはDirectX 9.0 SDKをインストールして、インクルードパスやライブラリパスを設定しておく必要があります。

「LibGame」フォルダには、DirectXに関する処理をまとめたソースコードが収録されています。DirectXを使ったゲームの基礎となる初期化処理や2D/3Dグラフィックの表示処理、ジョイスティックやキーボードからの入力処理などに関するプログラミングの参考にしてください。

「sht」フォルダには、シューティングゲームに特化したソースコードを収録しました。各アルゴリズム固有の処理もここにまとめられています。

▼「sht」フォルダ

▼「LibGame」フォルダ

注意　デモプログラムならびにプロジェクトファイル一式を利用するにあたっては、付録CD-ROMのreadme.txtをよく読み、注意事項に従ってください。

シューティングゲーム アルゴリズム マニアックス

松浦健一郎 著

サポートページのお知らせ

本書に関する情報をインターネット上でも公開しています。URLは以下のとおりです。

●ソフトバンクパブリッシング書籍のページ
http://store.sbpnet.jp/

●『シューティングゲームアルゴリズムマニアックス』のページ
http://www.Cmagazine.jp/books/stg/

●著者ホームページ
http://cgi32.plala.or.jp/higpen/gate.shtml

はじめに

本書は「シューティングゲームの仕組み」を解説する本です。ゲームプログラミングの入門書としてももちろん使えますが、普通の入門書とは少し毛色が違います。本書は次のような目的にお勧めです。

・シューティングゲームの中身がどんな仕組みで動いているのかを知って、ゲームをもっと楽しみたい
・シューティングゲームを自分で作りたい
・学校の課題でシューティングゲームを作ることになったが、いったいどこから手をつけていいのかわからない
・誘導レーザーや触手といったカッコいいしかけの実現方法が知りたい
・シューティングゲームの「技術カタログ」や「ネタ集」を求めている
・シューティングゲームについて熱く語るための、叩き台になる本がほしい
・昔遊んだり作ったりしたシューティングゲームのことを思いだしながら、懐かしい気持ちにひたりたい
・とにかくシューティングゲームが好きだ

本書はいわゆる入門書ではないので、どこから読んでもかまいませんし、好きなところだけひろい読みしても大丈夫です。図解を中心にしているので、まずは図だけでもパラパラと漫画を読むような気楽な気分でお読みください。一方で、各技法をシンプルにまとめたプログラムも多数掲載しているので、シューティングゲームの自作にも役立てていただけることと思います。

プレイヤーの方にとってもゲームデザイナーの方にとっても、本書がシューティングゲームの世界をより深く楽しむためのガイドブックになれば幸いです。

2004年夏　松浦 健一郎

Contents

Stage01 序章 Introduction

シューティングゲームの基本構成……002
シューティングゲームの進行……003
シューティングゲームを作るには……004
プラットフォームと開発環境……005
　お勧めの開発環境……007
サンプルプログラム……007

Stage02 弾 Bullet

狙い撃ち弾……010
　弾の動き……010
　弾の速度……011
　距離ゼロの場合の弾速計算……013
DDAを使って弾を動かす……015
固定小数点数を使って弾を動かす……018
　固定小数点数を使った計算……018
　シフト演算と固定小数点数……020
　固定小数点数を使った弾の移動……021
方向弾……023
テーブルを使った方向弾……024
DDAを使った方向弾……026
　整数型のみで処理を行う……027
弾の消去……029
弾の当たり判定処理……031
　当たり判定の設定……031
n-way弾……033
　n-way弾の発射……034
円形弾……036
分裂弾……038
誘導弾……039
　弾の誘導をあまくする……040
　旋回方向の選択……041
誘導レーザー……046

誘導レーザーの描画……049
誘導レーザーの当たり判定……050
ミサイル……052
加速弾……054
落下弾……056
回転弾……058
回転半径を変化させる……059
狙い撃ち弾+回転弾……061
回転の中心となる弾の消去……061
渦巻き弾……062
誘導弾のアレンジ……063
停止する誘導弾……064
逃げる誘導弾……064
直進するビーム……065
攻撃の予兆……066
安全地帯とその対策……067
弾の動きとランダム性……070

Stage03 自機 MyShip

自機の移動……074
8方向に自機を移動させる……074
自機の移動可能範囲……076
ロールの表示……077
3Dでのロール表示……079
斜め移動の速さ……080
アナログスティックを使った移動……082
アナログ入力値と「あそび」……082
入力値から移動量への変換……083
ワープ移動……085
ボタンによる自機のスピード調節……088
アイテムによる自機のスピード調節……089
合体する自機……091
地上を歩く自機……093
変形する自機……096
水中の移動……098
ゲージを使ったパワーアップ……101
オプション……104
オプションのアルゴリズム……104
オプションのプログラム……106
バリア……108
ボタンで張るバリア……110

Stage04 武器 Weapon

基本のショット操作……114
連射……116
溜め撃ち……117
連射と溜め撃ちの共存（セミオート連射）……119
　連射処理……119
　溜め撃ち処理……121
ボタンを離して溜める溜め撃ち……124
連射とレーザーの共存……126
ロックショット……129
コマンドショット……131
　コマンド入力のゆらぎ……132
　入力時間が異なるコマンドの扱い……134
ショットの移動……137
ショットの当たり判定処理……138
敵との距離によるショットの威力の違い……140
照準を使った爆撃……141
ロックオンレーザー……144
　レーザーの動き……145
地形に沿って飛ぶミサイル……148
　ミサイルの当たり判定……149
地形で反射するショット……151
地形で反射するレーザー……153
武器の切り替え……155
方向切り替えによる全方位射撃……157

Stage05 特殊攻撃 Special Attack

ボム……162
ボムとゲームバランス……164
近接攻撃……165
誘爆……167
アイテムによる特殊攻撃……169
無敵状態……172
バーサーク状態……174
味方をつかんで投げる……178
敵をつかまえて投げる……181
　アームの動き……182
　投げつけられた敵の動き……184

敵をつかまえて味方にする……185
敵につかまった自機を取り返してパワーアップする……188
味方に接近してショットを強化する……190
味方がいる方向に強いショットを撃つ……192
味方に当てたショットを強化する……193
自機の色を切り替えて弾を吸収する……195
敵弾をショットとして跳ね返す……197
レーザー同士をぶつける……200
敵弾に自機をかすらせてパワーアップする……202
　同じ弾に何度かすれるか……203
弾や敵の動きをスローにする……205
自由に動かせる照準……207
自機と照準を同時に動かす……210

Stage06 敵 Enemy

破壊できる敵……214
破壊できない敵……216
敵の出現と消失……218
空中に現れる敵……220
敵の速さと硬さのバランス……223
固定砲台……224
軌道を描いて飛ぶ敵……226
　軌道データを作っておく……226
　軌道データとプログラムの組み合わせ……228
敵の編隊……229
　座標の保存……231
　編隊の出現……232
複雑な形の当たり判定……234
ボスキャラの構造……237
ボスキャラの行動……239
ボスキャラの分離と合体……241
ボスキャラの変形……244
触手……246
　触手の先端を動かす……247
　触手の中間部分を動かす……248
多関節……250
　多関節のアルゴリズム（角度の決定）……252
　多関節のアルゴリズム（位置の計算）……254
撃ち返し……257

Stage07 背景 Scroll

横画面と縦画面……262
スクロール……265
背景の表示……270
多重スクロール……272
星のスクロール……274
強制縦スクロール+限定横スクロール……276
強制横スクロール+任意縦スクロール……279
回転……281
高速スクロール……283
　自機に力をかける……284
プレイヤーによるスクロール速度の調節……286

Stage08 システム System

桁数の多いスコア……290
　初期化……291
　加算(多倍長+整数)……292
　加算(多倍長+多倍長)……292
　乗算(多倍長×整数)……293
スコアに関するデザインの指針……297
リプレイ……299
乱数……302
　乱数の種……302
　乱数の生成方法……302
難易度……303
ゲームを快適にするために……306

Bonus Stage 付録 Appendix

Appendix 1　デモプログラム一覧……312
Appendix 2　引用ゲーム一覧……323
Appendix 3　索引……336

序章 Introduction

敵や弾をかいくぐり、ショットやレーザーを撃ってスコアを稼ぐ。この単純にして熱いプレイがシューティングゲームの醍醐味です。ほとんどのものが「8方向スティック+ショットボタン+特殊攻撃ボタン」というシンプルな操作系ながらも、緊張感あふれる弾よけや凝った演出を楽しめることが、シューティングゲームの魅力だといえます。

ハードウェアの進化にともなってグラフィックやサウンドの品質は向上しましたが、「よけて、撃つ」というシューティングゲームの本質は昔からほとんど変わっていません。この基本ルールのなかで、それぞれのゲームは「弾幕の美しさ」「点稼ぎの面白さ」「高い難易度」「魅力ある演出」といったさまざまな工夫を凝らしています。

シューティングゲームの基本構成

シューティングゲームに必要な要素はそう多くはありません。ほとんどはFig.1-1に示すような要素から構成されています。

自機

プレイヤーが操作する機体です。戦闘機や宇宙船を模していることが多いのですが、人や動物の姿をしていることもあります。

敵

自機に攻撃をしかけてくる相手です。多くの敵は自機の放つ武器を使って破壊することができます。

武器

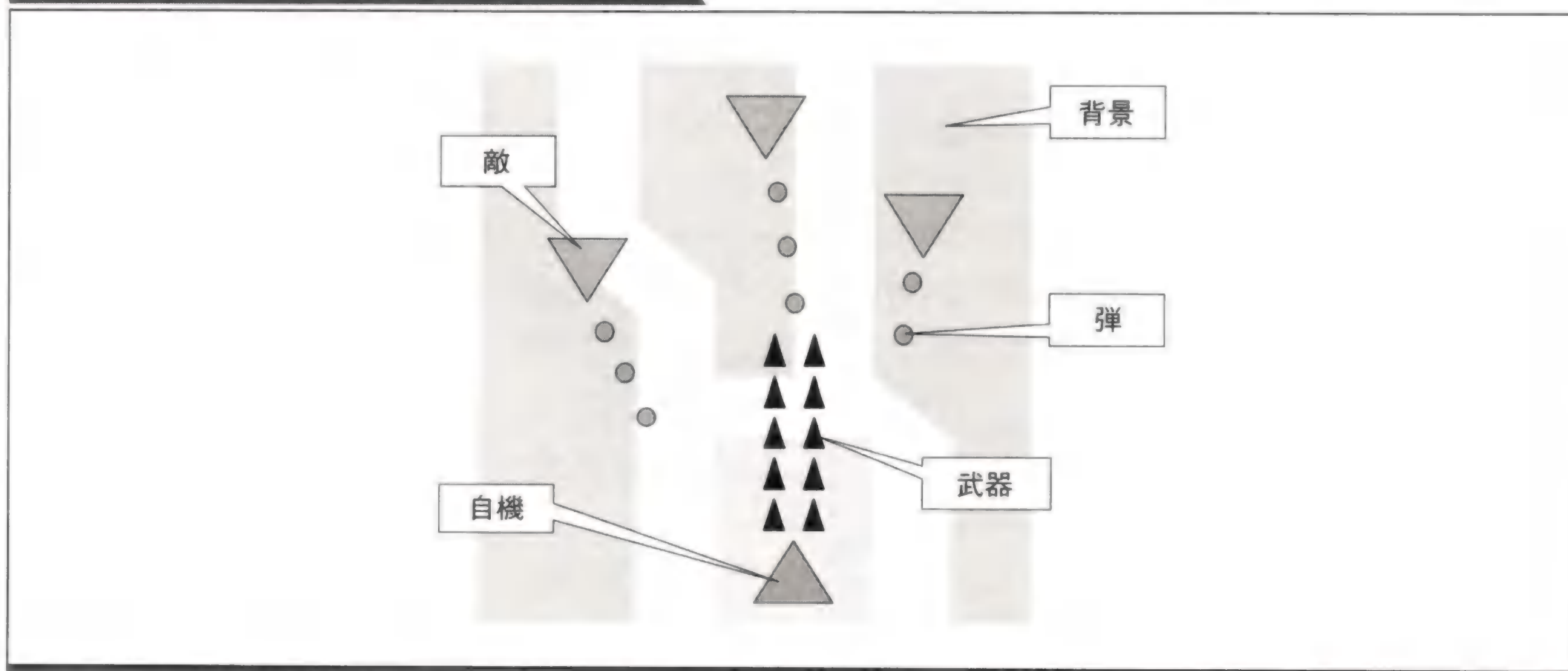
自機が敵に向かって放つ攻撃です。攻撃を敵に当てると破壊する（ダメージを与える）ことができます。

弾

敵が自機に向けて放ってくる攻撃です。多くの場合、弾を破壊することはできません（逆にいえば、破壊できる弾もあるということです）。弾に接触すると自機は破壊されます。

背景

自機、敵、弾の背後に表示される地面や宇宙空間などです。多くのゲームでは背景がスクロ

Fig. 1-1 シューティングゲームの基本構成

ール（ある方向に流れること）します。

■当たり判定

物体同士が接触したかどうかの判定です。自機と敵、自機と弾が接触したときには自機が破壊されます。敵と武器が接触したときには敵を破壊することができます。

■スコア

得点です。多くの場合、敵を破壊したりアイテムを拾ったりするとスコアが入ります。

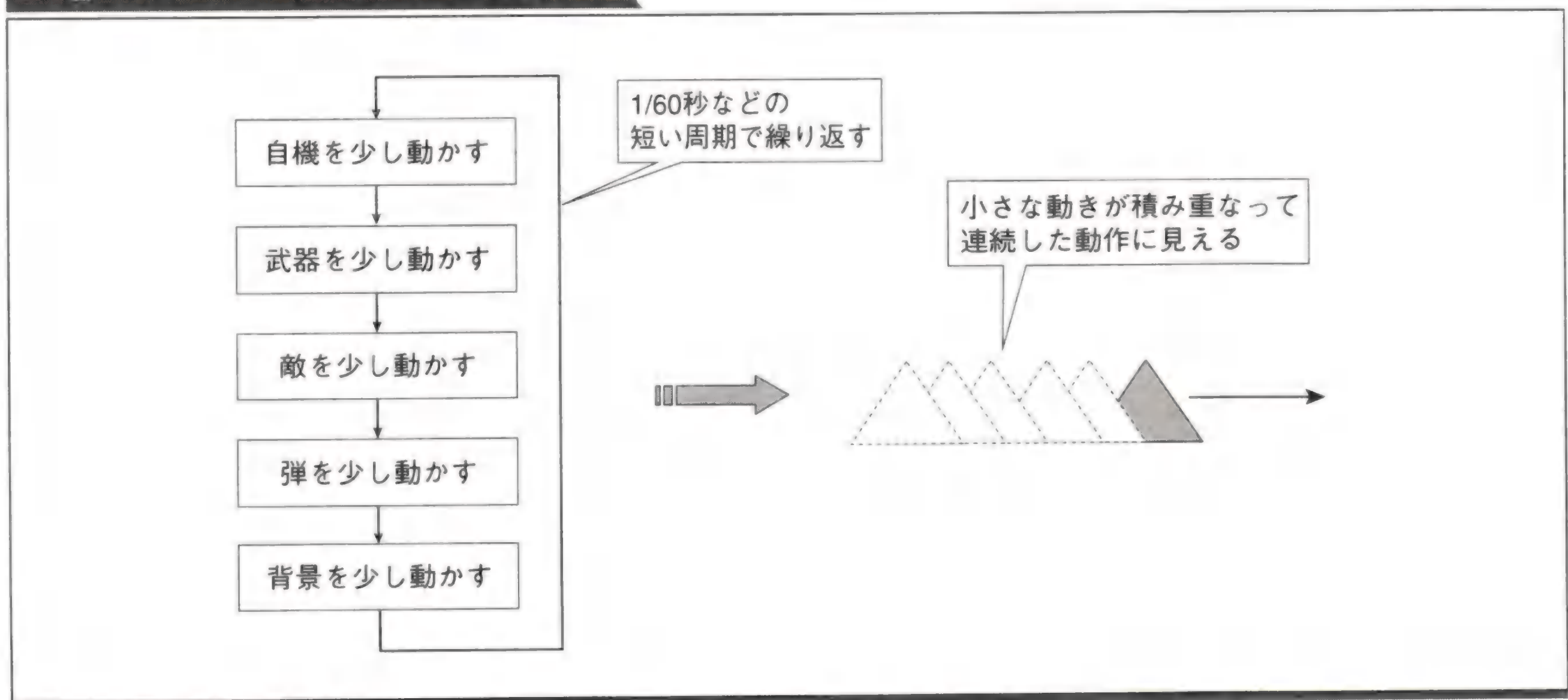
※

ゲームによってシステム（ルール）が異なり、自機と敵が接触しても破壊されない、逆にスコアが追加される、というケースもあります。ゲームごとに細かい差はありますが、基本的には上記のような要素の組み合わせでシューティングゲームは成り立っています。

本書では「弾」「自機」「武器」「敵」「背景」の順に解説を進めます。「当たり判定」については各章の関連する部分で、「スコア」については「システム」の章で解説します。

シューティングゲームの進行

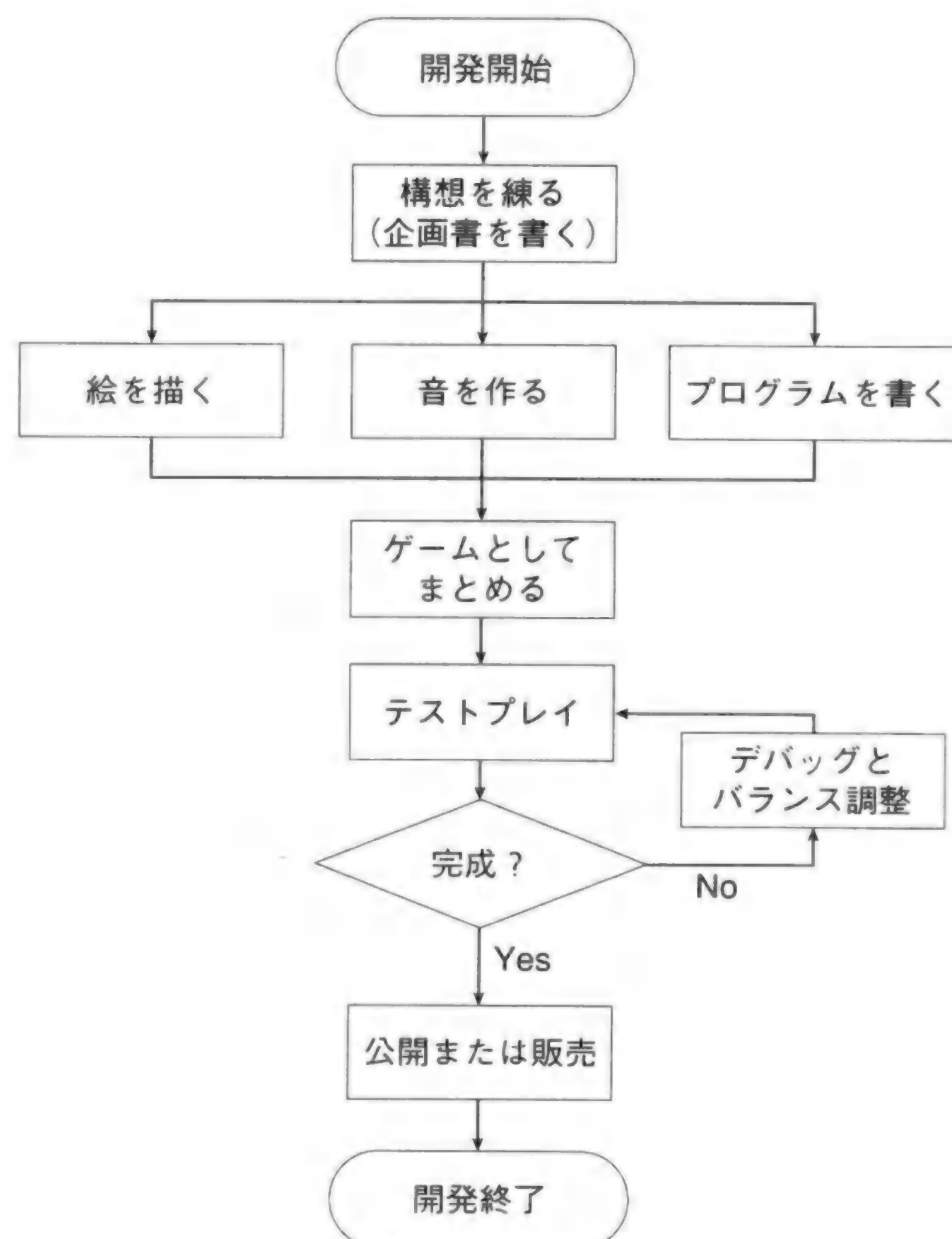
シューティングゲームの画面上では自機や弾などがとぎれなくなめらかに動きますが、実はこの動きは小さな動きの積み重ねからできています（Fig. 1-2）。たとえば、画面の左端から右端まで自機が動く場合、最初は画面幅の1/100だけ動かし、次にまた1/100動かし……、といった処理を100回積み重ねて、連続した動きを作り出します。

Fig. 1-2 シューティングゲームの進行

この小さな動きは非常に短い周期で繰り返されます。多くのゲームでは、これを画面表示の周期に合わせます。たとえば、家庭用TVやPC用ディスプレイの一般的な周期は1/60秒なので、ゲームも1/60秒周期で進行します。表示の周期とゲームの周期を合わせると、アニメーションをなめらかに表示することができるからです。

また、武器の発射や当たり判定などの処理も、この動きに合わせて行います。画面の書き換えと同じ周期で、さまざまな計算処理を行います。

シューティングゲームを作るには

シューティングゲームを作る手順は、趣味で作る場合も仕事で作る場合も、あるいは学校の課題として作る場合も、基本はそれほど変わりません (Fig. 1-3)。開発に複数の人間がかかわる場合には、作業を分担し、いくつかの作業を並行して進めます。

Fig. 1-3 シューティングゲームを作る手順

開発の最初はゲームの構想を練ることから始まります。仕事の場合には企画書や仕様書を書いたり、開発予算を確保したりする必要もあります。基本的な構想がある程度できたら、絵、音、プログラムの作成に入ります。データやプログラムを作りながら、さらに構想をふくらませていくこともあるでしょう。

よいゲームを作るために大事なのは、テストプレイを重ねることです。単にゲームとして動けばよいというだけではなく、時間の許すかぎりテストプレイ、デバッグ、ゲームバランス調整を繰り返します。

プラットフォームと開発環境

シューティングゲームのプログラムを作るために必要な準備について解説します。まずはゲームを動かすプラットフォームを選ばなくてはなりません。プラットフォームには主に次のような選択肢があります。

PC

個人がゲームを作る際にもっとも手軽に使えるのはPCです。選択できる言語の種類が多く、ライブラリもよく整備されています。PC向けのシューティングゲームの市販品に関しては、アーケードゲームの移植版が低価格で発売されているものの、家庭用ゲーム機に比べると本数は少ないようです。

携帯電話/携帯端末

iモード、Palm、Pocket PC、Zaurusなどのプログラムは個人でも開発することができます。最近では端末の性能が向上したので、ひと昔前のアーケードゲーム程度のものも動くようになりました。

携帯ゲーム機

ワンダースワンやPIECEについては、個人がプログラミングを楽しむための開発キットが販売されています。ゲームボーイアドバンスについては正式な開発キットはリリースされていませんが、エミュレータなどを用いた開発は個人でも行われています。

家庭用ゲーム機/アーケード基板

仕事でゲームを開発する場合にはこのどちらかになることが多いでしょう。どちらもPCとは違ってハードウェアの仕様にばらつきがないので、ハードウェアの性能を限界まで生かしたゲームを作るのに適した環境です。

※

趣味や同人、あるいは学校の課題でゲームを作る場合には、プラットフォームとしてPCを

使うことが多いでしょう。PCでゲームを作る場合には、多くの開発環境のなかから好きなものを選ぶことができます。

ここでは、主な開発環境とその特徴を紹介します。

C/C++とDirectX

Windowsを使う場合にはもっとも一般的な選択です。C/C++コンパイラは各社から発売されています。DirectXとの相性のよさではMicrosoft Visual C++に軍配が上がりますが、無償で利用できるBorland C++ Compilerも魅力的です。

C/C++とOpenGL

DirectXよりもOpenGLのほうを好むプログラマも少なくありません。OpenGLの利点は、DirectXほどには仕様の変化が急激ではないことと、Windows以外の環境でも利用できることです。

Java

最近は専門学校や大学でもJavaを教えるところが多いようで、Javaは確実に普及しています。JavaはC/C++よりも速度の点では不利ですが、よいライブラリを使えばJavaでも十分な速度のゲームは作れます。

Delphi (Object Pascal)

Delphiはボーランド社の開発環境で、Object Pascal言語をベースにしています。Delphiの個人向けバージョンは無償で利用できることが魅力です。DelphiからDirectXを使うためのライブラリも公開されています(「Quadruple D」など)。

HSP

HSP (Hot Soup Processor) は、手軽に使えることで人気を集めているスクリプト言語システムです。ゲームを作りたいけれども、C/C++やDirectXを勉強するのは敷居が高いという方にもお勧めできます。

シューティングゲーム作成ツール

プログラミング言語やライブラリを勉強しなくても、絵を描いたり簡単な設定をしたりするだけでゲームが作れるツールもあります。「シューティングツクール」シリーズ (アスキー) などが有名どころです。

※

開発環境やライブラリは個人の好みで選ぶのがいちばんです。本書のプログラム例は、PCをプラットフォームとし、C/C++でDirectXを使うことを前提としています。とはいえ、どの言語でゲームを作る場合にも基本的な考え方や手法は変わりません。本書の図を見ながらリストに手を加えれば、ほかの言語で動くプログラムを作るのも難しくないでしょう。

■ お勧めの開発環境

著者の個人的なお勧めは「C/C++とDirectX」の組み合わせか、もしくは「C/C++とOpenGL」の組み合わせです。もっと勉強が少なくてすむ組み合わせはたくさんありますが、世の中のゲームの大部分がC/C++で書かれていることを考えると、C/C++を覚えておくことには大いに意味があります。

特に「将来はゲームプログラマになろう」と考えている方は、C/C++をしっかり学んでおくべきです。C/C++はゲーム以外のプログラミングにも大いに役立つので、C/C++を身につけておけば、おそらくプログラマとして仕事がなくなることはないでしょう。Javaも悪くはありませんが、まだまだC/C++のほうが用途は広いと思われます。欲をいえば、C/C++とJavaの両方を学んでおけば完璧です。

● サンプルプログラム

本書で紹介するさまざまなアルゴリズムが実際に動いているところを確認できるように、ゲームふうのサンプルを用意し、付録CD-ROMに収録してあります。実際に自機を操作して、弾をよけたり敵を倒したりする行うことができます。

また、本書はシューティングゲームに登場するさまざまなアルゴリズムを解説することを主題としているため、シューティングゲームそのものの作り方に関する解説は省略させていただきます。しかし、ご安心ください。付録CD-ROMに収録したサンプルは、表示からキー操作などシューティングゲームの基本的な処理を備えています。このサンプルのソースコードが、シューティングゲームの基本的な部分を作成する参考になってくれることでしょう。

また、本書内で掲載しているソースコードは、各アルゴリズムの中核となる部分を抜粋したものです。ソースコード単体ではコンパイルや実行できないものもあります。コンパイルや実行が可能な状態のソースコードは、サンプル内の各アルゴリズムに対応する部分をご参照ください。

Stage 02 ▶▶

弾 Bullet

敵が放つ弾は、おそらくシューティングゲームにおいてもっとも大事な要素でしょう。シューティングゲーム好きにとっては、ただひたすら弾幕をよけ続けるだけでも、けっこう面白く遊べてしまうものです。ここ最近の流行として、「弾速は遅めに、自機の当たり判定は小さくして、画面を埋めつくすほどの弾幕を張る」というのがあります。昔のハードウェアでは難しかったのですが、現在の高速なハードウェアならば、膨大な数の弾を同時に動かすことができます。弾幕で美しいパターンを描き出すゲームも多く、そういったゲームはプレイせずに画面を眺めているだけでも楽しめます。

弾のプログラムを作るうえで基本となるのは、「狙い撃ち弾」(自機の方向に飛ぶ弾)と「方向弾」(自由な方向に飛ぶ弾)という2種類の弾です。複雑な弾幕も、たいていはこれら2種類の弾を組み合わせれば作ることができます。本章では、基本的な弾の処理に加えて、誘導レーザーや誘導ミサイルといった特殊な弾のアルゴリズムも解説します。

狙い撃ち弾

敵が放つ弾の種類はさまざまですが、もっともオーソドックスなのは自機の方向に向かって飛んでくる弾で、ほとんどのシューティングゲームで見かけます。本書ではこの弾を「狙い撃ち弾」と呼ぶことにします（Fig. 2-1）。

■ 弾の動き

狙い撃ち弾の動きを細かく分析してみましょう。Fig. 2-2は発射された弾の軌跡です。

狙い撃ち弾は敵と同じ座標からスタートして、発射時の自機の座標を目指します。自機がまったく動かなければ弾は命中しますが、たいていのプレイヤーは回避行動をとるでしょう。その場合にはFig. 2-3のように、弾は元と同じ進行方向を保ったまま直進します。画面の外に出たら、その弾は消えて終わりです。

Fig. 2-1 狙い撃ち弾

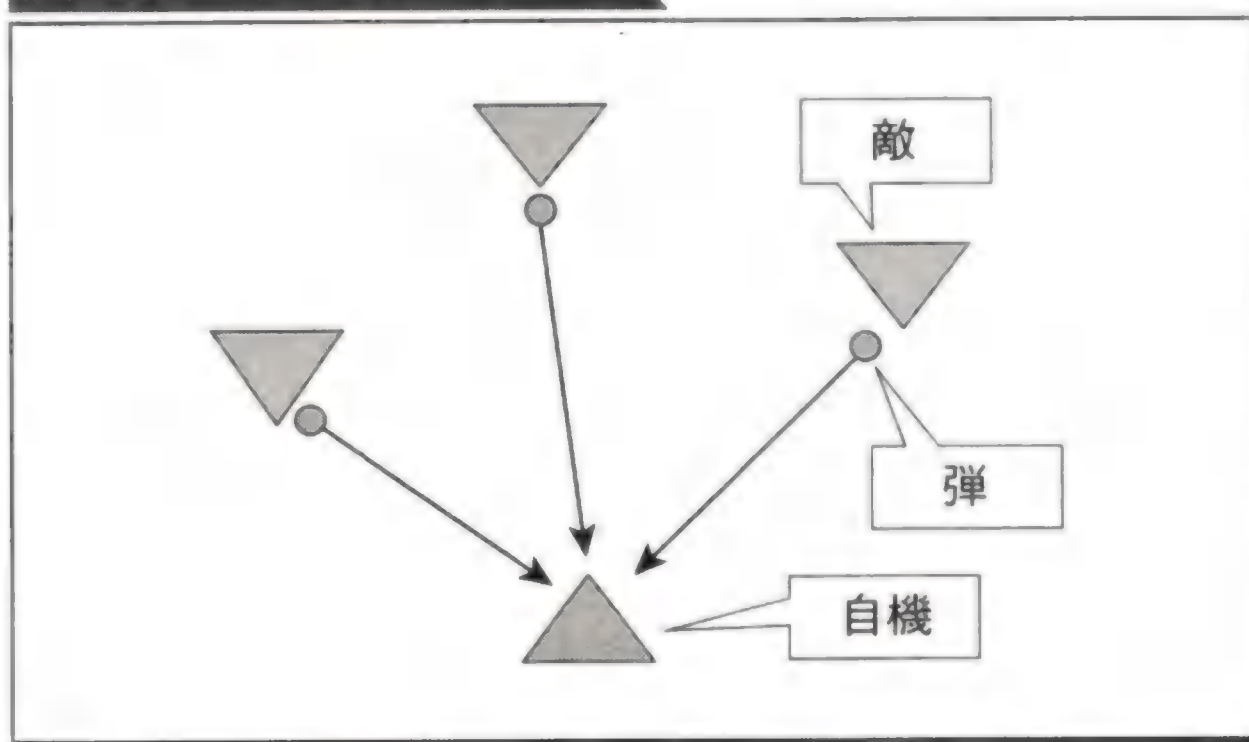



Fig. 2-2 弾の動きの分析

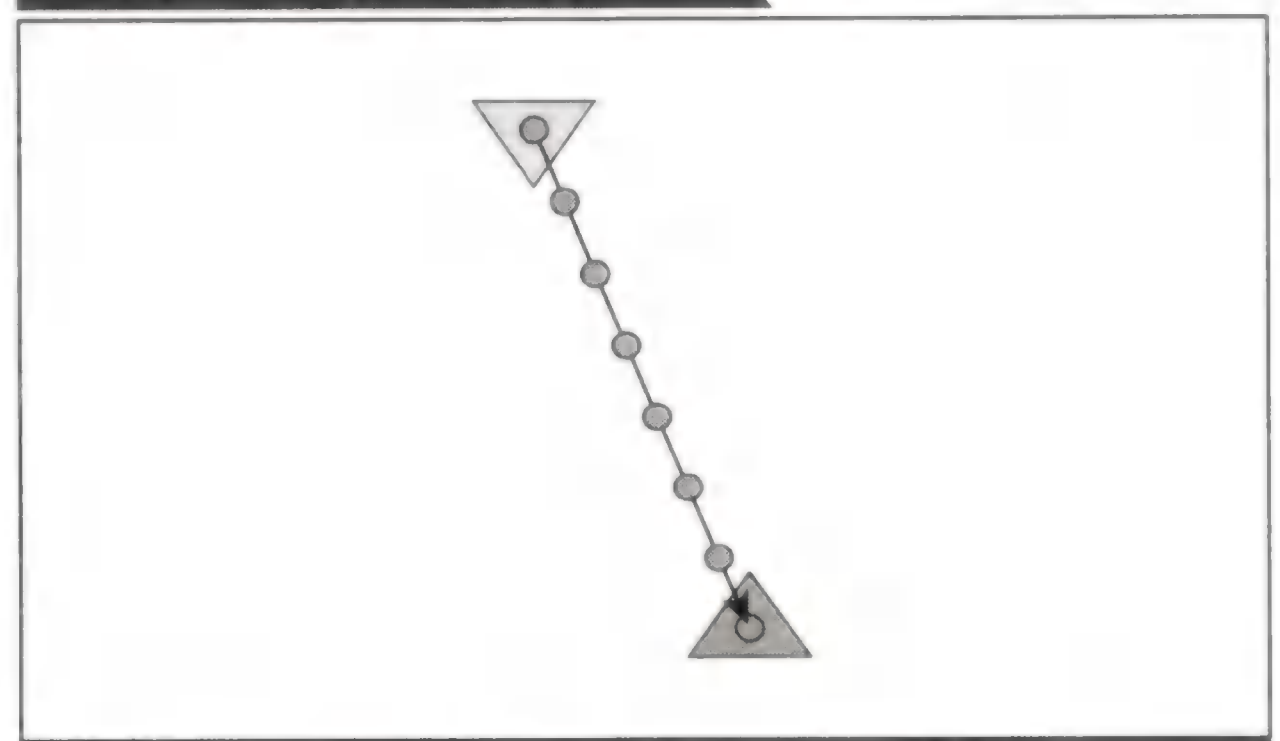
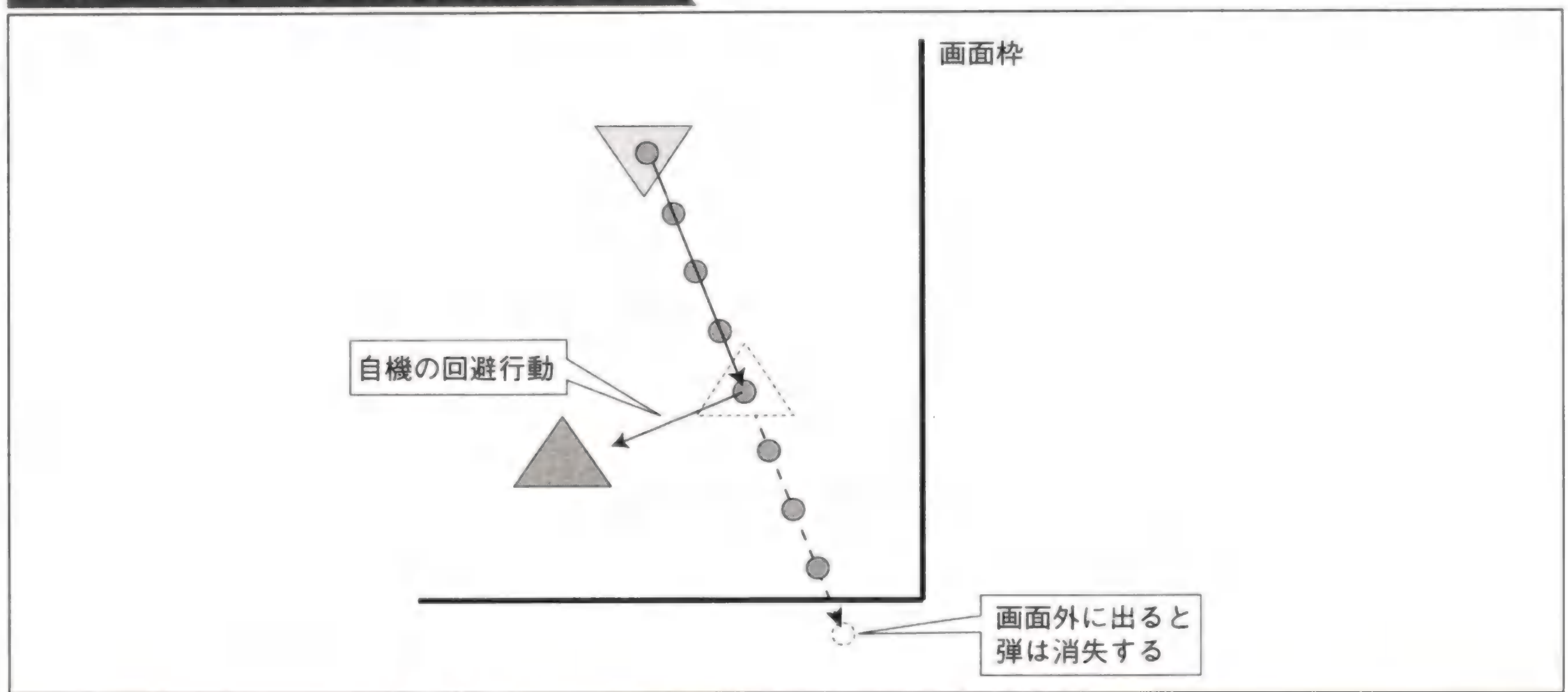



Fig. 2-3 自機に回避されたあとの弾の動き

弾の速度

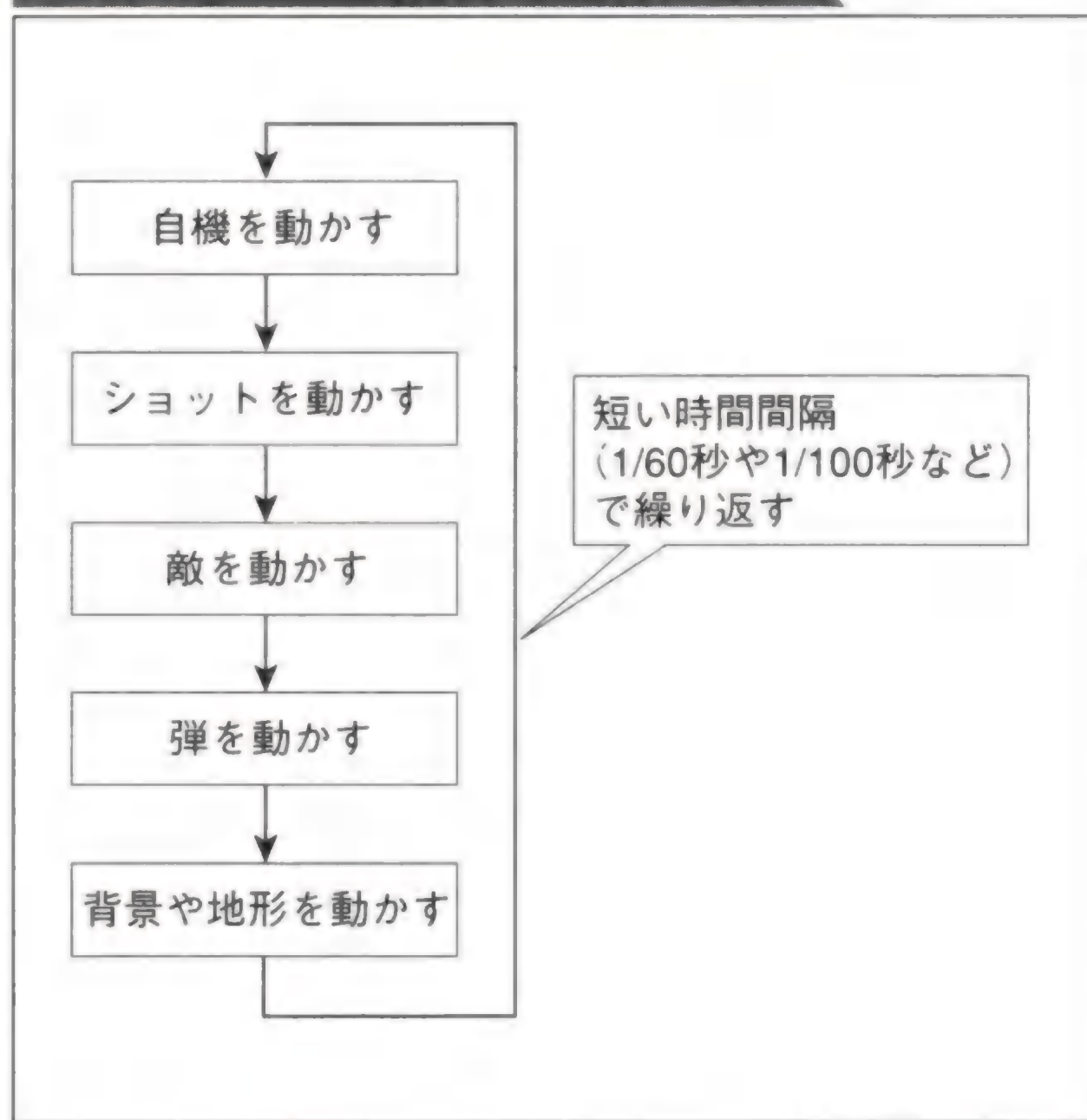
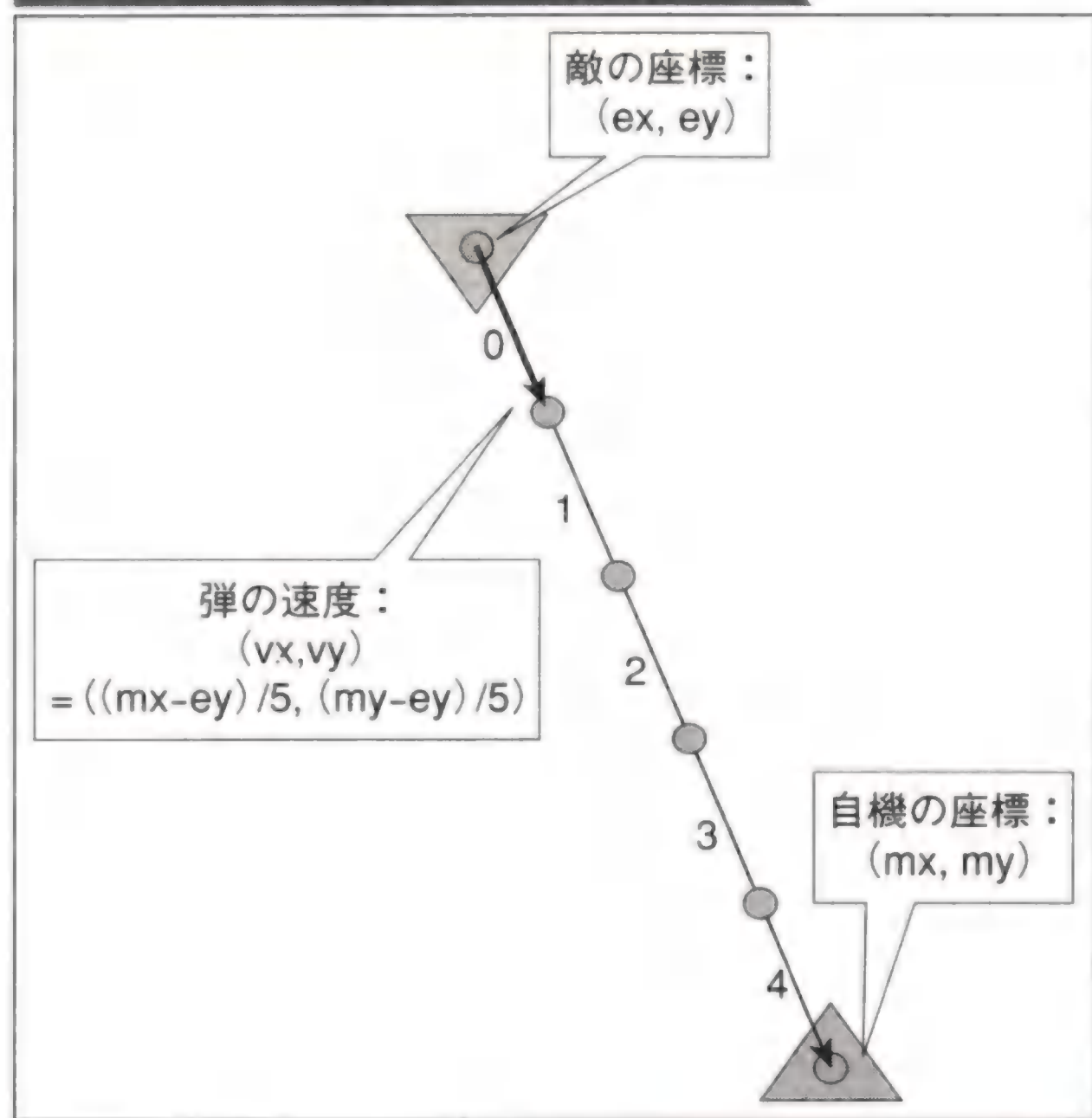
シューティングゲームのプログラムは、Fig. 2-4のような仕組みで動きます。プログラムは自機や弾などを短い周期（1/60秒など）で少しずつ動かします。そして、この小さな動きを積み重ねることによって、連続的な動きを作り出します。

自機に向かってくる弾の場合、Fig. 2-4に示す1周期に弾を動かす量、つまり弾の速度はFig. 2-5のように計算します。ここでは2次元（2Dグラフィック）の場合を考えますが、3次元（3Dグラフィック）の場合でも考え方は同じです。

Fig. 2-4 プログラムが動く仕組み

自機を動かす
ショットを動かす
敵を動かす
弾を動かす
背景や地形を動かす
短い時間間隔（1/60秒や1/100秒など）で繰り返す

Fig. 2-5 弾の速度を求める方法

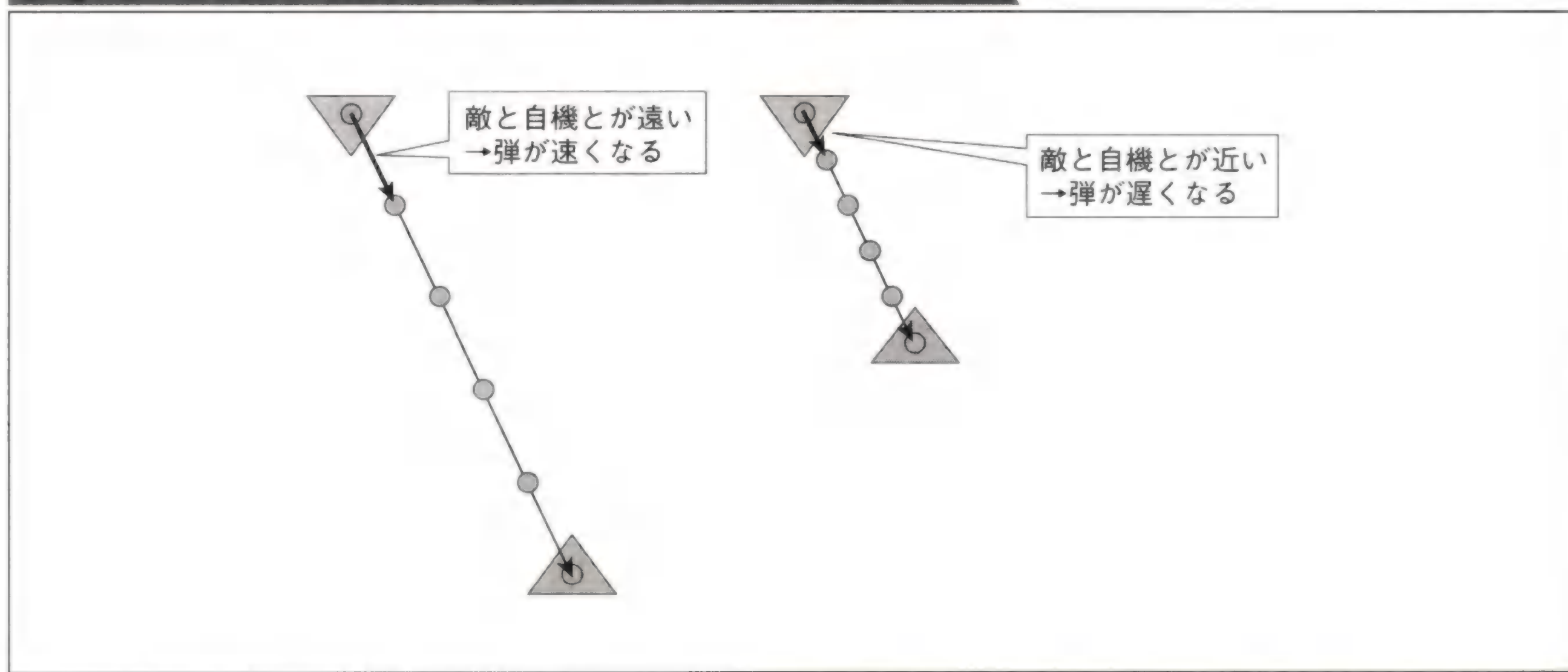
敵の座標：(ex, ey)
弾の速度：(vx,vy) =((mx-ey)/5, (my-ey)/5)
自機の座標：(mx, my)
0 1 2 3 4

敵の座標を（ex, ey）、自機の座標を（mx, my）、弾の速度を（vx, vy）とします。弾は敵の座標（ex, ey）からスタートします。すると、Fig. 2-5のように5回の移動で自機の座標（mx, my）に到達するための速度は次の式で求められます。

vx=(mx-ex)/5
vy=(my-ey)/5

これをより一般的にして、n回の移動で到達するための速度は、次の式で求まることになります。

vx=(mx-ex)/n
vy=(my-ey)/n

ところで、この方法では敵と自機との位置関係によって弾の速さが違ってしまいます（Fig. 2-6）。敵と自機とが遠いときには弾が速くなり、近いときには遅くなります。これはこれで効果としては面白いかもしれませんが、普通は弾の速さは一定にします。

Fig. 2-6 敵と自機との位置関係によって弾の速さが変わってしまう

弾の速度を一定にするには、Fig. 2-7のような方法を使います。ここでは弾の速さをspeedとしました。まず、敵から自機までの距離dを求めます。

d=sqrt ((mx-ex) * (mx-ex) + (my-ey) * (my-ey))

sqrtは平方根を求める関数です。C/C++の場合にはmath.hで定義されています。

次に、弾の速度を求めます。Fig. 2-5の場合に似ていますが、今度は移動回数nで割るのではなく、距離dで割ったあとに速さspeedを掛けます。

vx= (mx-ex) /d*speed
vy= (my-ey) /d*speed

これで敵と自機の位置関係にかかわらず、弾の速さは一定値（speed）になります。数学ふうにいえば、Fig. 2-7の計算は「弾の速度ベクトル（vx, vy）の長さを一定値speedにする」ということです。

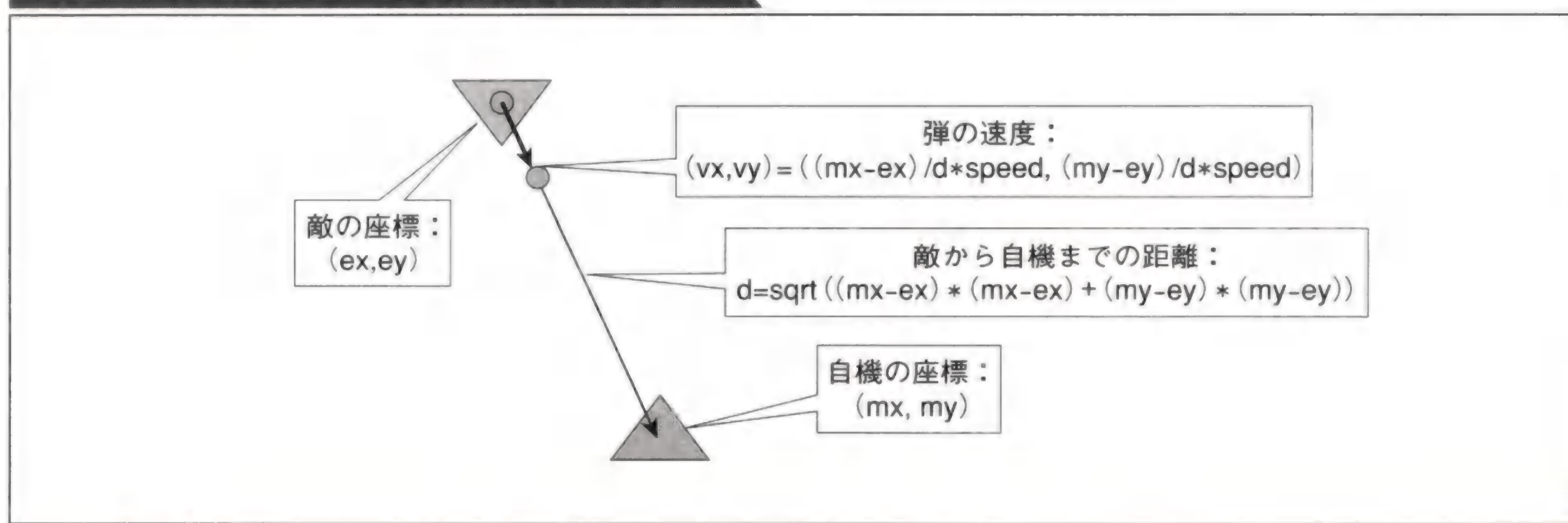
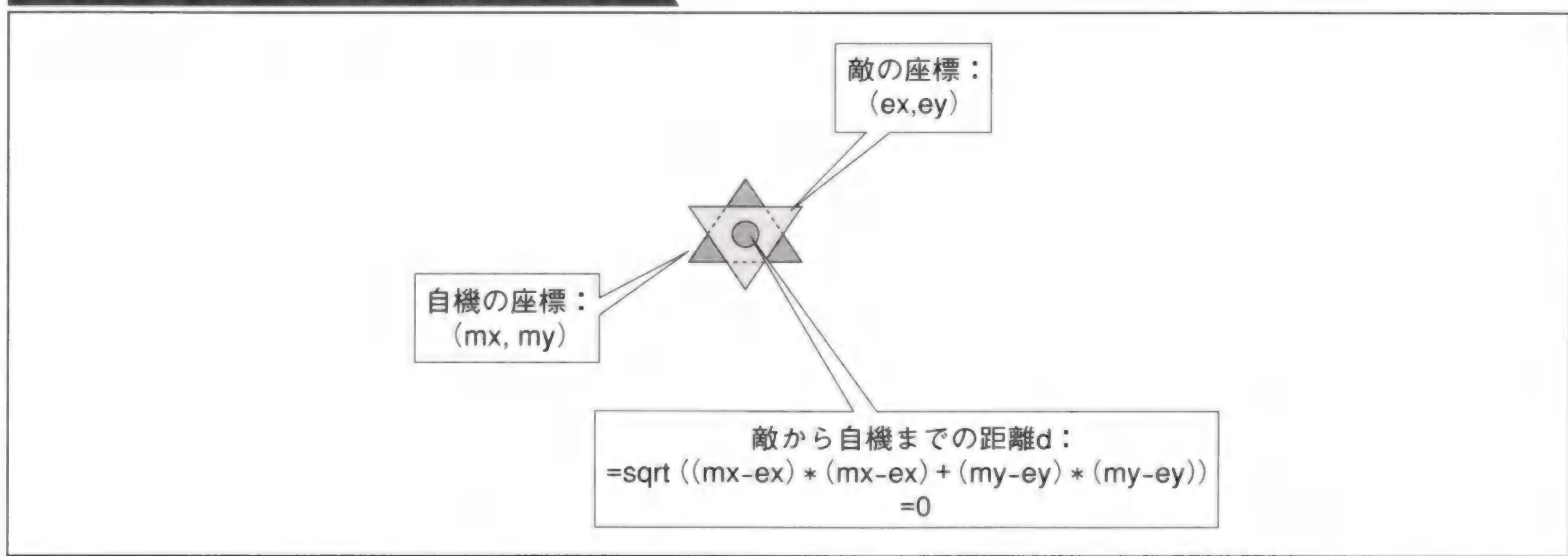
Fig. 2-7 弾の速さを一定にするための計算方法

■ 距離ゼロの場合の弾速計算

Fig. 2-7の計算には1つ注意点があります。ここでは距離dで割り算を行いますが、dが0のときには割り算ができません。0で割ることはできないからです。割り算ができないと、弾の速度を決めることができません。

距離が0になるのは、Fig. 2-8のように敵と自機とが重なっている状況です。敵との衝突がミスになるゲームならば、このように敵と自機とが重なることはないかもしれません。しかし、敵との衝突がミスにならないゲームや、自機の耐久度がゲージ制になっているゲームなどでは、Fig. 2-8のような状況が頻繁に発生します。

Fig. 2-8 敵と自機とが重なっている状況

このように敵と自機とが重なっている状況（dが0になる状況）には、たとえば次のように対処します。

- **弾を発射しない**
- **特定の方向（下方向など）に弾を発射する**
- **ランダムな方向に弾を発射する**

距離dの値を求めたあとにdが0かどうかを判定して、0の場合には上記のように特別な処理を行います。

たとえば、自機と敵の間に距離がある場合は狙い撃ちをし、距離がゼロの場合は決まった方向（下方向）に弾を発射する処理は、List 2-1のようなプログラムになります。

一方、弾を移動させるプログラムはList 2-2のようになります。List 2-1で求めた速度（vx, vy）を弾の座標（x, y）に加算するだけです。

サンプル

● 狙い撃ち弾　→ P. 312
● 狙い撃ち弾2　→ P. 312

List 2-1 狙い撃ち弾の初期化

```
#include <math.h>

void InitAimingBullet(
    float mx, float my,    // 自機の座標
    float ex, float ey,    // 敵の座標
    float speed,           // 弾の速さ
    float& x, float& y,    // 弾の座標
    float& vx, float& vy   // 弾の速度
) {
    // 弾の座標を設定する
    x=ex; y=ey;

    // 目標までの距離dを求める
    float d=sqrt((mx-ex)*(mx-ex)+(my-ey)*(my-ey));

    // 速さが一定値speedになるように速度(vx, vy)を求める：
    // 目標までの距離dが0のときには下方向に発射する。
    if (d) {
        vx=(mx-ex)/d*speed;
        vy=(my-ey)/d*speed;
    } else {
        vx=0;
        vy=speed;
    }
```

List 2-2 狙い撃ち弾の移動

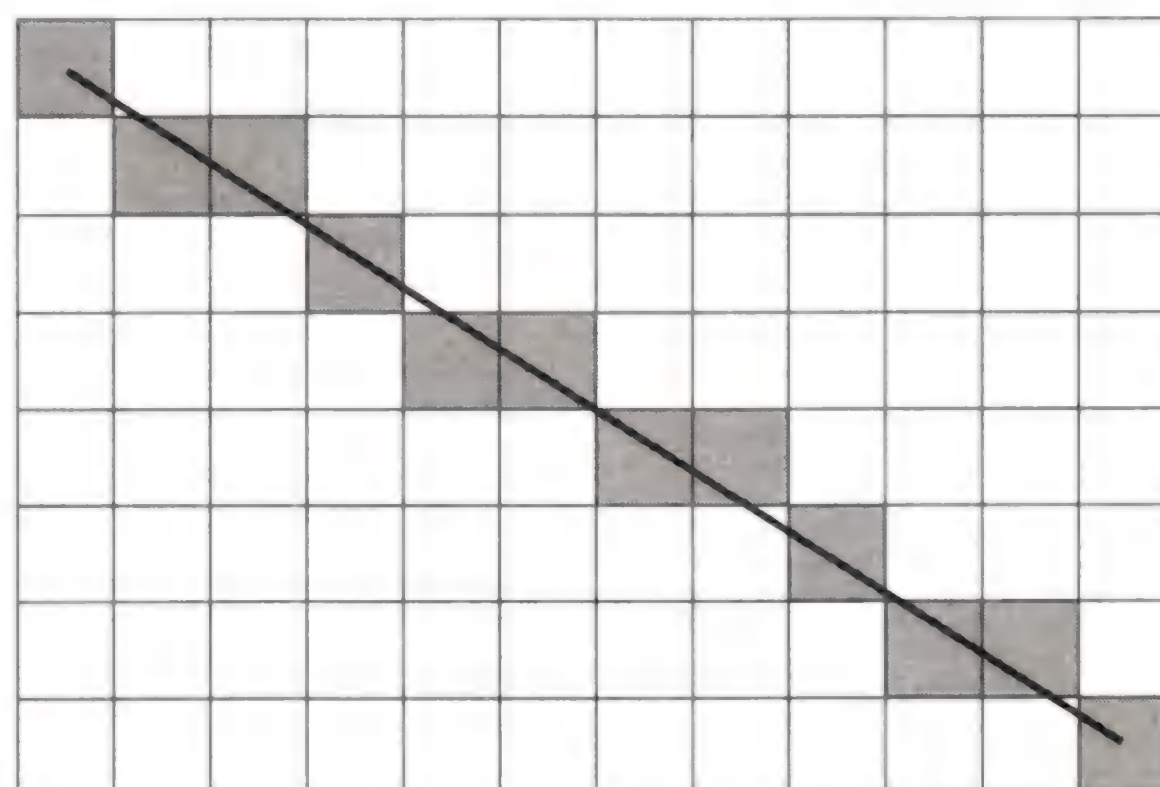
```
void MoveAimingBullet(
    float& x, float& y,   // 弾の座標
    float vx, float vy    // 弾の速度
) {
    // 弾の座標(x, y)に速度(vx, vy)を加える
    x+=vx;
    y+=vy;
}
```

DDAを使って弾を動かす

狙い撃ち弾のアルゴリズムを解説しましたが、この方法では、プログラムにfloatのような実数型を使う必要があります。最近のPCは高速なので、実数型を使っても処理速度上の問題はほとんどありませんが、低速な携帯端末や古いアーケードゲーム基板などでは問題になることがあります。

たとえば、携帯電話などで使うJava実行環境（Java 2 Platform,Micro Edition：J2ME）は実数型をサポートしていません。また、アーケードゲーム基盤のなかには、実数演算機能を持たない古いプロセッサが載ったものもあります。そのような環境でシューティングゲームを作るときには、実数型を使わないプログラムにする必要があります。

実数型を使わないで弾を移動させるには、「DDA（Digital Differential Analyzer：デジタル微分解析器）」を使う方法があります。DDAというのはCGで線分（ライン）を描くときに使う手法です（Fig. 2-9）。コンピュータのピクセル（画素）数はかぎられているので、まっすぐに見える線分でも、実際にはFig. 2-9のような階段状の図形になっています。DDAは本来の線分にもっとも近い階段状のピクセルを整数演算だけで高速に求めるアルゴリズムです。

Fig. 2-9 DDAによる線分の描画

DDAを応用すると、整数演算だけで弾を好きな方向に飛ばすことができます。DDAを言葉で説明するのはなかなか難しいので、DDAを使って弾を動かすプログラムをList 2-3、2-4にまとめました。ポイントはList 2-4にある2つのforループです。ここでは次のような処理を行います。

- **X方向とY方向のうち目標までの座標の差分が大きいほうは毎回移動させる**
- **差分が短いほうについては誤差diffが蓄積したときだけ移動させる**

プログラムをにらみながらじっくり考えると、これは「目標に対するX方向の差分：目標に対するY方向の差分」という比で弾をX方向およびY方向に動かすという処理だ、ということが

わかるでしょう。つまり、弾は目標に向かってほぼまっすぐに飛んでいくことになります。

List 2-3、2-4の方法の弱点は、弾の速さにばらつきが出ることです。ここでは弾の速さをspeedとしましたが、弾が水平や垂直に飛ぶときには速さがspeedに等しくなるのに対して、斜めに飛ぶときにはもう少し速くなります。斜め45度に飛ぶときがもっとも速くなり、このときは水平方向や垂直方向に飛ぶときに比べて約1.4倍（sin45°倍）の速さになります。こういった速さの差さえ気にならなければ、この方法は整数演算だけで弾を飛ばすときにとても役立ちます。

サンプル

狙い撃ち弾（DDA） → P. 312

List 2-3 DDAを使った狙い撃ち弾の初期化

```
void InitAimingBulletDDA(
    int mx, int my,     // 自機の座標
    int ex, int ey,     // 敵の座標
    int& x, int& y,     // 弾の座標
    int& vx, int& vy,   // 弾の移動方向
    int& dx, int& dy,   // X方向とY方向の差分
    int& diff           // 誤差
) {
    // 弾の座標を設定する
    x=ex; y=ey;

    // 弾の移動方向(vx, vy)を求める：値は1または-1
    vx=mx>ex?1:-1;
    vy=my>ey?1:-1;

    // 目標に対するX方向とY方向の差分の絶対値(dx, dy)を求める
    dx=mx>=ex?mx-ex:ex-mx;
    dy=my>=ey?my-ey:ey-my;

    // 誤差diff：dx>=dyのときはdx/2、dx<dyのときはdy/2とする
    diff=dx>=dy?dx/2:dy/2;
}
```

List 2-4 DDAを使った狙い撃ち弾の移動

```
void MoveAimingBulletDDA(
    int& x, int& y,    // 弾の座標
    int vx, int vy,    // 弾の移動方向
    int dx, int dy,    // X方向とY方向の差分
    int& diff,         // 誤差
    int speed          // 弾の速さ
) {
    // 移動距離のX方向が長いときの処理
    if (dx>=dy) {
        for (int i=0; i<speed; i++) {

            // X方向には毎回移動させる
            x+=vx;

            // Y方向には誤差が蓄積したときだけ移動させる
            diff+=dy;
            if (diff>=dx) {
                diff-=dx;
                y+=vy;
            }
        }
    }

    // 移動距離のY方向が長いときの処理
    else {
        for (int i=0; i<speed; i++) {

            // Y方向には毎回移動させる
            y+=vy;

            // X方向には誤差が蓄積したときだけ移動させる
            diff+=dx;
            if (diff>=dy) {
                diff-=dy;
                x+=vx;
            }
        }
    }
}
```

固定小数点数を使って弾を動かす

実数型（float）を使わないで弾を動かすもう1つの方法は、固定小数点数を使うことです。固定小数点数では、小数に決まった数（たとえば100や10000など）を掛けることによって、小数を整数に変えて扱います。

たとえば、0.37という小数値を考えます。これはintなどの整数型では表すことができませんが、100を掛けて37にすれば整数として扱えます。同じように0.21や3.94といった小数も21や394といった整数に変えることができます。このように固定小数点数というのは、すべての数に決まった数を掛けることによって、小数を整数で表す手法です。

固定小数点数を使った計算

固定小数点数を使った計算をしてみましょう。たとえば次の足し算を考えます（Fig. 2-10）。

0.37+3.94

この計算結果は4.31です。これを固定小数点数で行うと次の計算になります（桁上げ用に100を掛けた場合です）。

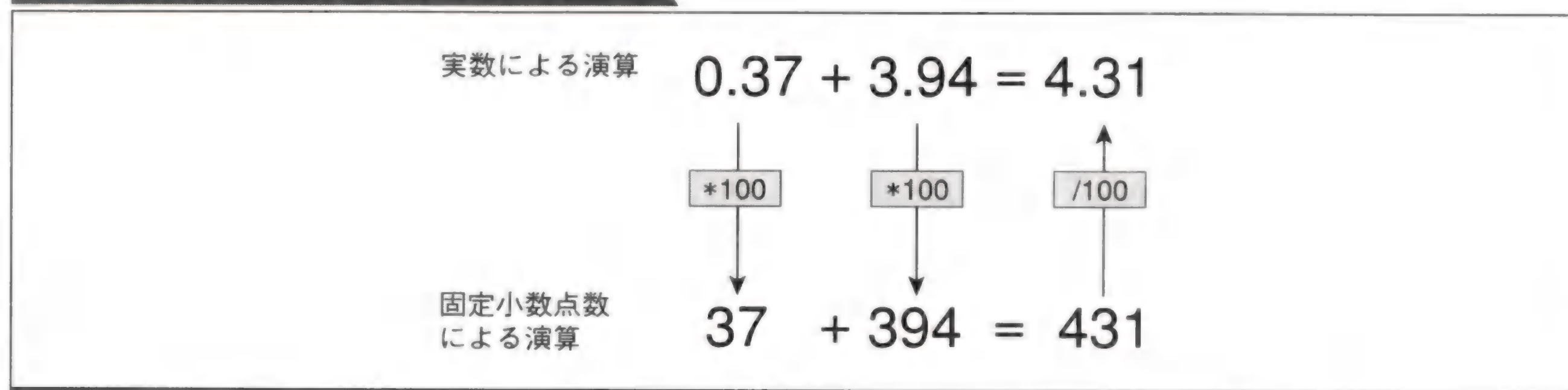
37+394

この計算結果は431ですから、100で割れば本来の計算結果である4.31になります。こうすることで、整数のみで計算を行うことができます。引き算についても足し算と同様に計算することができます。

Fig. 2-10 固定小数点数を使った足し算

実数による演算　0.37 + 3.94 = 4.31

*100　*100　/100

固定小数点数による演算　37 + 394 = 431

次に、掛け算の場合を考えます（Fig. 2-11）。

0.37*3.94

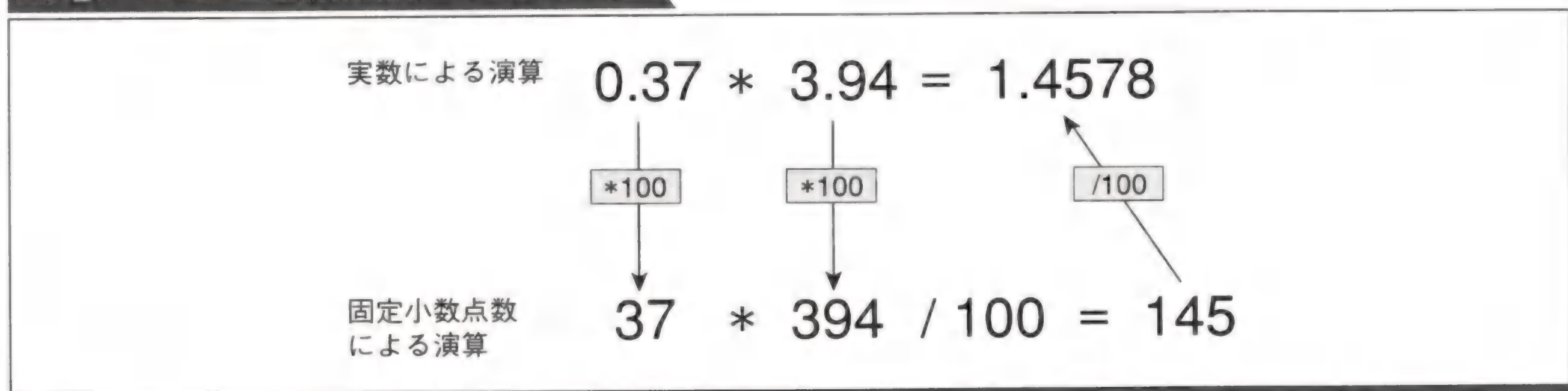
実数による計算結果は1.4578です。固定小数点数の場合には次の計算になります（桁上げ用に100を掛けた場合です）。

37*394/100

このように掛け算では、計算結果を桁上げ用の数値（ここでは100）で割ります。これは掛け算の左辺（37）と右辺（394）にそれぞれ100を掛けているため、掛け算を行うと100*100=10000が掛かったことになってしまうからです。

(0.37*100) * (3.94*100)
= (0.37*3.94) * (100*100)
=0.37*3.94*10000

固定小数点数による計算結果は145となり、100で割ると本来の計算結果である1.4578にほぼ等しくなります。固定小数点数ではこのように計算誤差が出ることが多いので、注意が必要です。

Fig. 2-11 固定小数点数を使った掛け算

最後に割り算について考えます（Fig. 2-12）。

0.37/3.94

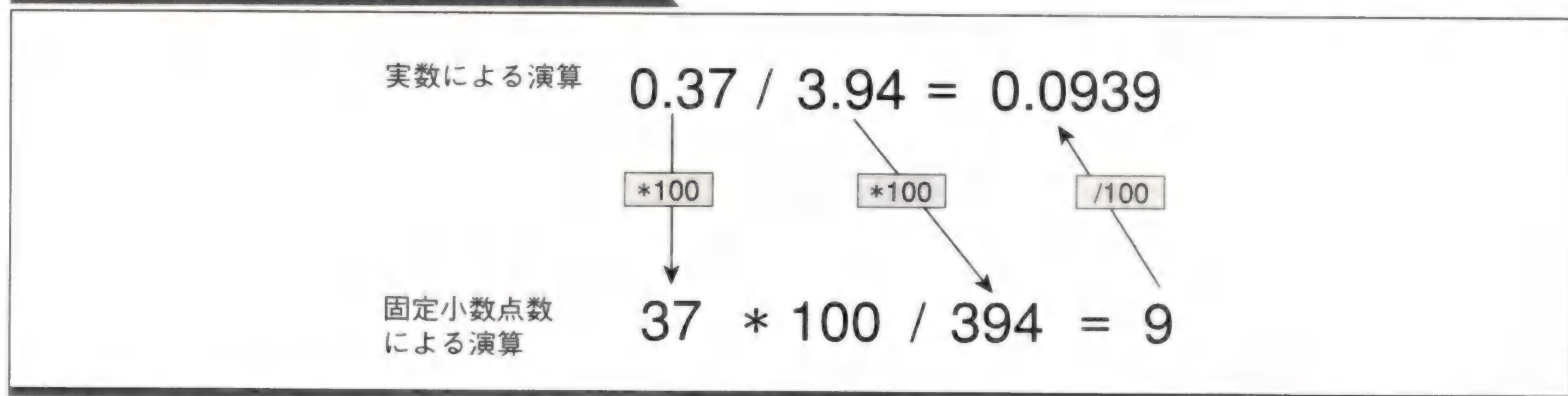
計算結果は約0.0939です。固定小数点数の場合には次の計算になります（桁上げ用に100を掛けた場合です）。

37*100/394

このように割り算では、割られる数に桁上げ用の数値（ここでは100）を掛けてから割ります。これは掛け算の場合とは逆の理由で、割り算を行うと次のように桁上げ用の数値が打ち消されてしまうからです。

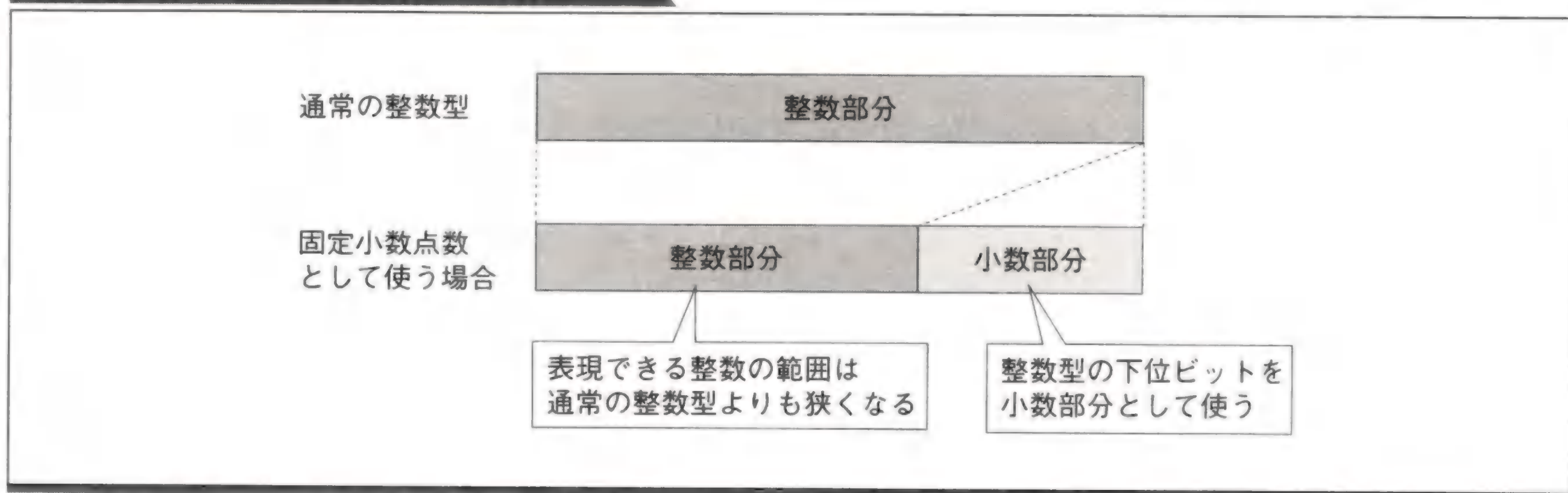
(0.37*100) / (3.94*100)
= (0.37/3.94) * (100/100)
=0.37/3.94

固定小数点数による計算結果は9となり、それを100で割ると、やはり誤差はあるものの実数の計算結果とほぼ一致します。

Fig. 2-12 固定小数点数を使った割り算

■ シフト演算と固定小数点数

固定小数点数を使った演算では100や10000を掛けてもかまわないのですが、シフト演算のほうが掛け算よりも高速なので、シフト演算を使うほうが一般的です。シフトを使う場合、整数型の下位ビットを小数部分に割り当てることになります（Fig. 2-13）。たとえば、一般的なint型は32ビットの幅がありますから、下位8ビットや下位16ビットを小数部分に割り当てます。

Fig. 2-13のように整数型の一部を小数部分に割り当てるので、通常の整数型に比べると整数部分が減ります。つまり、表現できる整数の範囲が狭くなります。たとえば、32ビットのint型の場合、すべてを整数として使った場合には-2,147,483,648から2,147,483,647までの範囲を表せますが、下位8ビットを小数部分として使った場合には-8,388,608から8,388,607までの範囲に縮まります。掛け算や割り算を行う場合には、値が有効な範囲からあふれないよう、特に注意しなければいけません。

Fig. 2-13 シフトを使った固定小数点数

ここでは値aとbをシフトを使った固定小数点数だとして、aとbとの間の計算を考えます。まず、足し算と引き算は次のようになります。

```
a+b
a-b
```

掛け算と割り算は、それぞれ次のようになります（下位8ビットを小数部分とする場合です）。

```
a*b>>8
(a<<8)/b
```

このように、固定小数点数の桁上げを行うには次のように左にシフトを、桁下げを行うには次のように右にシフトします。

```
a<<8
a>>8
```

■ 固定小数点数を使った弾の移動

実数型を使う場合、自機に向かってくる弾の速度を求める式は次のとおりでした（→ P. 12）。

```
d=sqrt((mx-ex)*(mx-ex)+(my-ey)*(my-ey))
vx=(mx-ex)/d*speed
vy=(my-ey)/d*speed
```

固定小数点数を使う場合、後半の2つの式は次のようになります。

```
vx=(mx-ex)*speed/d
vy=(my-ey)*speed/d
```

ここでは桁上げを8ビットとし、自機の座標（mx, my）と敵の座標（ex, ey）およびspeedはすでに桁上げされているものとしました。割り算を最後にしているのは、一般に固定小数点数ではそうしたほうが誤差が少なくなるからです。

ここで問題なのは、dを求める1番目の式です。この式ではsqrt（平方根）を使いますが、一般にsqrtは実数型のための関数なので、せっかく固定小数点数を使っても実数演算が混じってしまうことになります。これを回避するには次のような方法があります。

①開平法などで平方根を求める固定小数点数向けの関数を書く
②距離のかわりにX方向またはY方向の差分のうち大きなほうを使う

①は正攻法です。手間はかかりますが正確な値が得られます。②はdのかわりにmx-exまたはmy-eyを使う方法です。計算は簡単ですが、弾を撃つ角度によって最大で1.4倍程度の速さの差が出てしまいます。

②の方法で距離dを求めるとすると、弾を発射するプログラムはList 2-5のようになります。

List 2-6は弾を移動させるプログラムです。弾の座標（x, y）は桁上げされているものとしました。弾を表示する際には座標（x, y）を桁下げして（x>>8, y>>8）とする必要があります。

※

固定小数点数を使った処理では、精度が十分にあれば弾の速度がばらつくこともないので、実数型を使った場合との違いはわずかです。ただし、固定小数点数を使ったプログラミングには手間がかかります。最近のPCでシューティングゲームを作る場合には、実数型を使っても速度上の問題はほとんど起きないので、シンプルに実数型を使ってしまうのがよいでしょう。

サンプル

狙い打ち弾（固定小数点数） → P. 312

List 2-5 固定小数点数を使った狙い撃ち弾の初期化

```
void InitAimingBulletFP(
    int mx, int my,     // 自機の座標
    int ex, int ey,     // 敵の座標
    int& x, int& y,     // 弾の座標
    int& vx, int& vy,   // 弾の速度
    int speed           // 弾の速さ
) {
    // 弾の座標を設定する
    x=ex; y=ey;

    // 目標に対するX方向とY方向の差分の絶対値(dx, dy)を求める
    int dx=mx>=ex?mx-ex:ex-mx;
    int dy=my>=ey?my-ey:ey-my;

    // X方向とY方向の差分のうち長いほうを距離dとする
    int d=dx>=dy?dx:dy;

    // 速度(vx, vy)を求める
    vx=(mx-ex)*speed/d;
    vy=(my-ey)*speed/d;
}
```

List 2-6 固定小数点数を使った狙い撃ち弾の移動

```
void MoveAimingBulletFP(
    int& x, int& y,  // 弾の座標
    int vx, int vy   // 弾の速度
) {
    // 弾の座標(x, y)に速度(vx, vy)を加える
    x+=vx;
    y+=vy;

    // 弾を描く：
    // 画面に弾を描くときには座標を(x>>8, y>>8)とする。
    // 弾を描く具体的な処理はDraw関数で行うとする。
    Draw(x>>8, y>>8);
}
```

方向弾

狙い撃ち弾（→P. 10）とは違って、特に自機を狙うというわけではなく、自由な方向に向かって飛ぶ弾です。本書ではこのような弾を「方向弾」と呼ぶことにします。この弾はたとえば、周期的に回転する砲台があって、砲台が向いている方向に弾をまき散らすような場合に使います（Fig. 2-14）。

方向弾の速度はFig. 2-15のように求めます。弾の速度を（vx, vy）、弾の速さをspeed、発射角度をtheta（θ）とすると、弾を発射するプログラムはList 2-7のようになります。

弾を移動させる処理はList 2-8のように書きます。弾の座標（x, y）に速度（vx, vy）を加えるだけです。

サンプル

方向弾 →P. 312

Fig. 2-14 砲台が向いている方向に弾を飛ばす

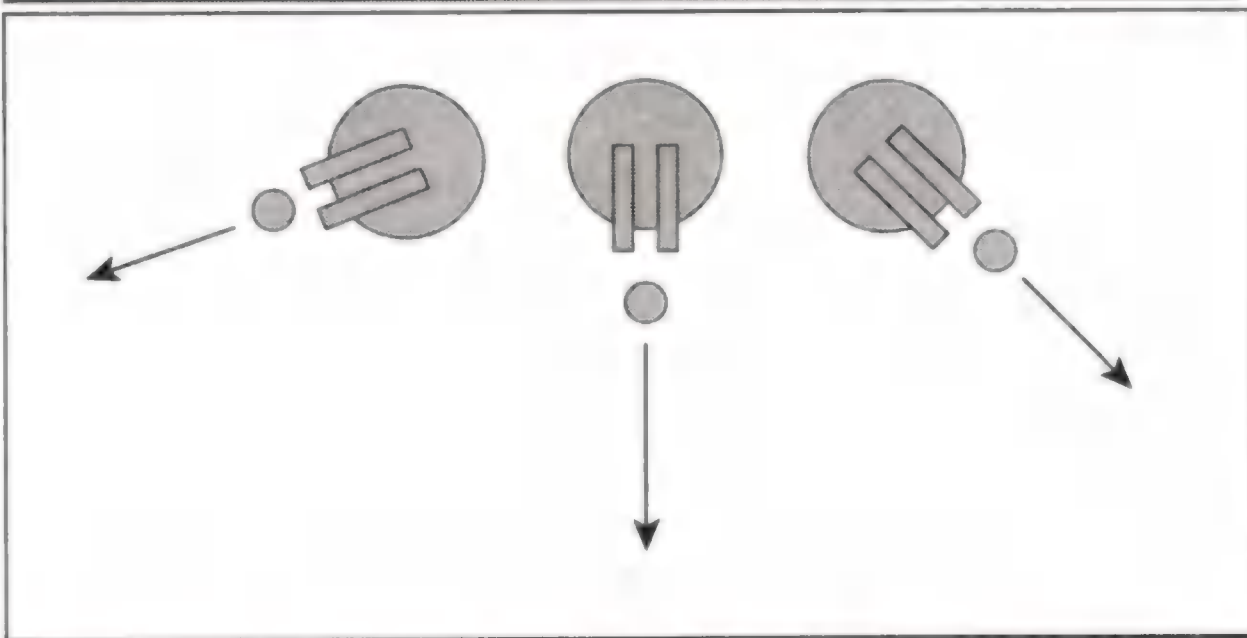
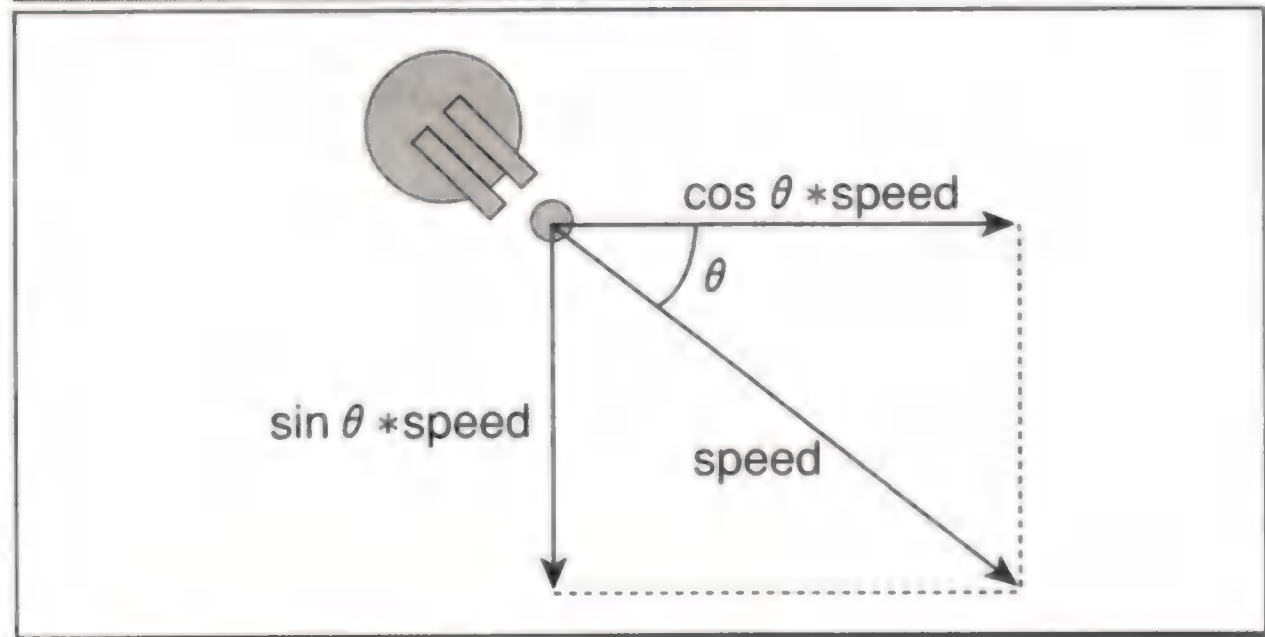



Fig. 2-15 方向弾の速度

List 2-7 方向弾の初期化

```
#include <math.h>

void InitDirectedBullet(
    float ex, float ey,     // 敵の座標
    float& x, float& y,     // 弾の座標
    float& vx, float& vy,   // 弾の速度
    float speed,            // 弾の速さ
    float theta             // 発射角度
) {
    // 弾の座標を設定する
    x=ex; y=ey;

    // 速さspeedで角度thetaの方向に飛ぶ弾の速度(vx, vy)を求める：
    // M_PIは円周率。
```

```
    vx=cos(M_PI/180*theta)*speed;
    vy=sin(M_PI/180*theta)*speed;
}
```

List 2-8 方向弾の移動

```
void MoveDirectedBullet(
    float& x, float& y,   // 弾の座標
    float vx, float vy    // 弾の速度
) {
    // 弾の座標(x, y)に速度(vx, vy)を加える
    x+=vx;
    y+=vy;
}
```

テーブルを使った方向弾

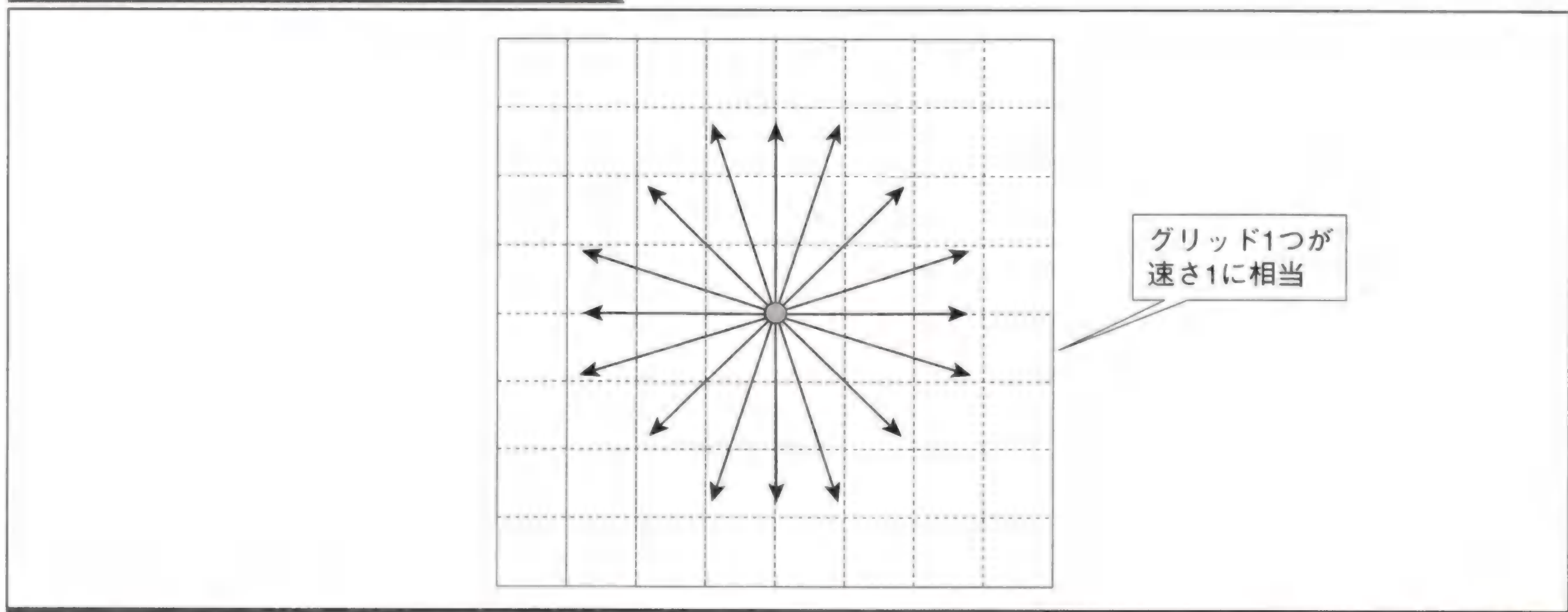
List 2-7と2-8では弾の処理に実数型（float型）を使いましたが、整数型（int型など）だけで方向弾を作る方法もあります。

Fig. 2-16は16方向に飛ぶ速さ3の弾の例です。各方向に対する弾の速度（vx, vy）は次のようになります（X軸の方向から右回りに）。

(3, 0) , (3, 1) , (2, 2) , (1, 3) ,
(0, 3) , (-1, 3) , (-2, 2) , (-3, 1) ,
(-3, 0) , (-3,-1) , (-2,-2) , (-1,-3) ,
(0,-3) , (1,-3) , (2,-2) , (3,-1)

このように、弾の速さに対する速度のテーブルを作っておくと、弾を発射するプログラムはList 2-9のようになります。一方、弾の移動は先ほどのList 2-8とまったく同じプログラムになります。

この方法の利点はシンプルなことです。問題点は、弾を飛ばせる方向に8方向や16方向といった制限があることと、方向によって微妙に弾の速度が違ってしまうこと、速さごとに速度のテーブルを用意する手間がかかることです。昔のゲームにはこの方法で弾を飛ばすものも多いですが、これでは最近のゲームのようにきめ細かな弾幕を作るのは難しいでしょう。実数型を使った場合に比べると、弾筋はかなりかぎられてしまいます。

Fig. 2-16 16方向に飛ぶ速さ3の弾

サンプル

テーブルを使った方向弾 → P. 312

List 2-9 16方向に飛ぶ速さ3の弾の発射

```
void InitDirectedBullet16_3(
    float& vx, float& vy,  // 弾の速度
    float theta            // 発射角度
) {
    // 速さ3に対する速度のテーブル
    static int v3[][2]={
        { 3, 0}, { 3, 1}, { 2, 2}, { 1, 3},
        { 0, 3}, {-1, 3}, {-2, 2}, {-3, 1},
        {-3, 0}, {-3,-1}, {-2,-2}, {-1,-3},
        { 0,-3}, { 1,-3}, { 2,-2}, { 3,-1}
    };

    // 角度theta (0°～360°) を16方向 (0～15) に変換する
    int dir=theta*16/360;

    // 弾の速度(vx, vy)をテーブルから求める
    vx=v3[dir][0];
    vy=v3[dir][1];
}
```

DDAを使った方向弾

テーブルを使った方向弾の問題点をクリアするには、先に説明したDDAを使うか、もしくは固定小数点数を使います。ここではDDAを使った方法を紹介しましょう。

DDAを使った狙い撃ち弾（→ P. 15）を応用すれば、方向弾も実現できます。Fig. 2-17のように「弾の周囲に自機がずらりと等間隔に並んだ状態」を考えると、「自由な方向に弾を飛ばす」というのは、「それぞれの方向に置かれた自機に向かって弾を飛ばす」のと同じことです。

そこで、弾を飛ばしたい方向に仮想的に自機を配置します（Fig. 2-18）。このときのポイントは、ある程度遠くに（たとえば距離100や1000などに）自機を配置することです。遠くに配置することによって、弾を飛ばす方向の精度が上がります。あとは配置した自機に向かって弾を発射すれば、DDAで自機に向かう弾を飛ばす処理とまったく同じになります。

List 2-10は自機を仮想的に配置するプログラムです。後半は狙い撃ち弾とまったく同じ処理になります。

Fig. 2-17 弾の周囲に自機が並んだ状態

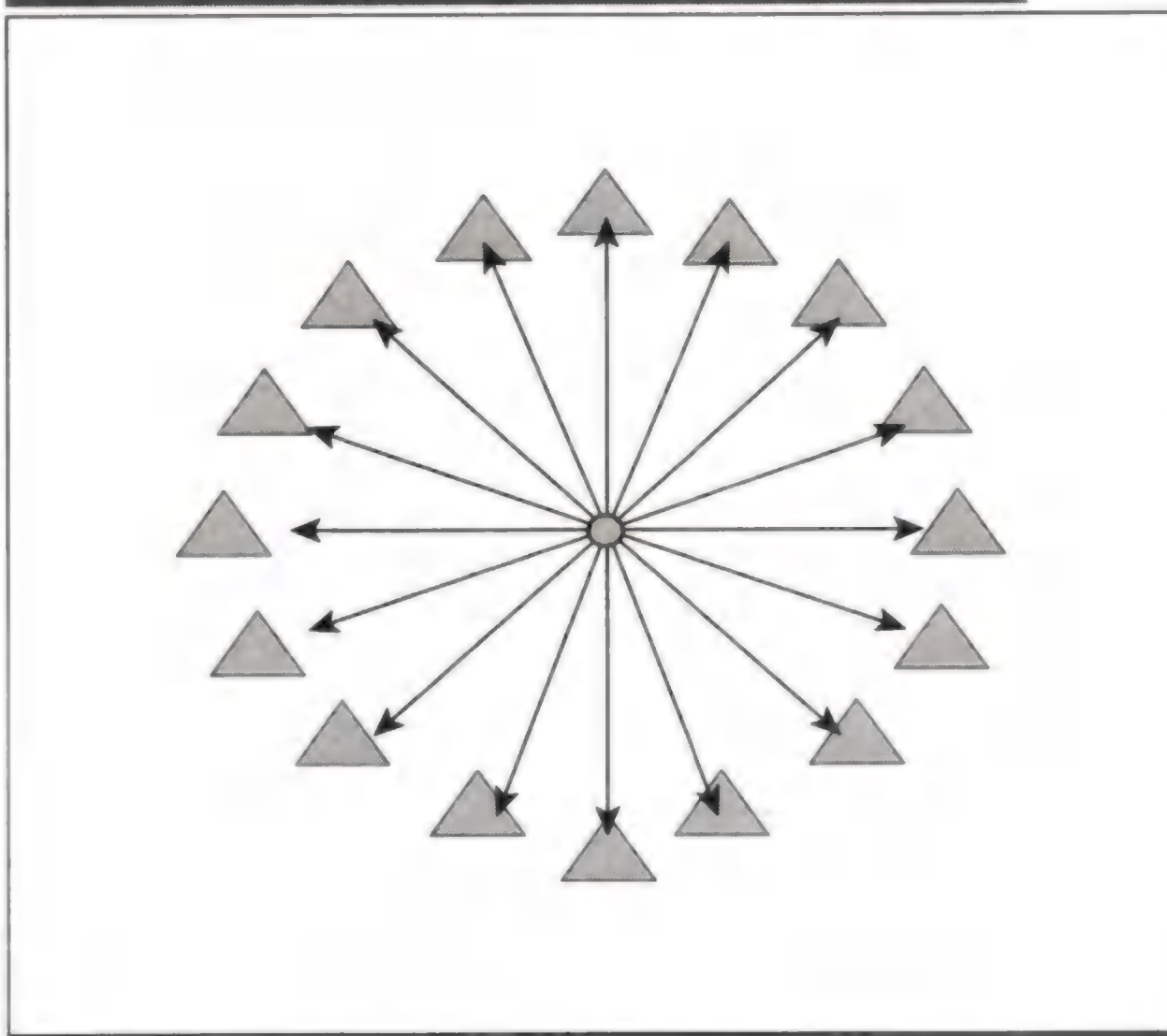
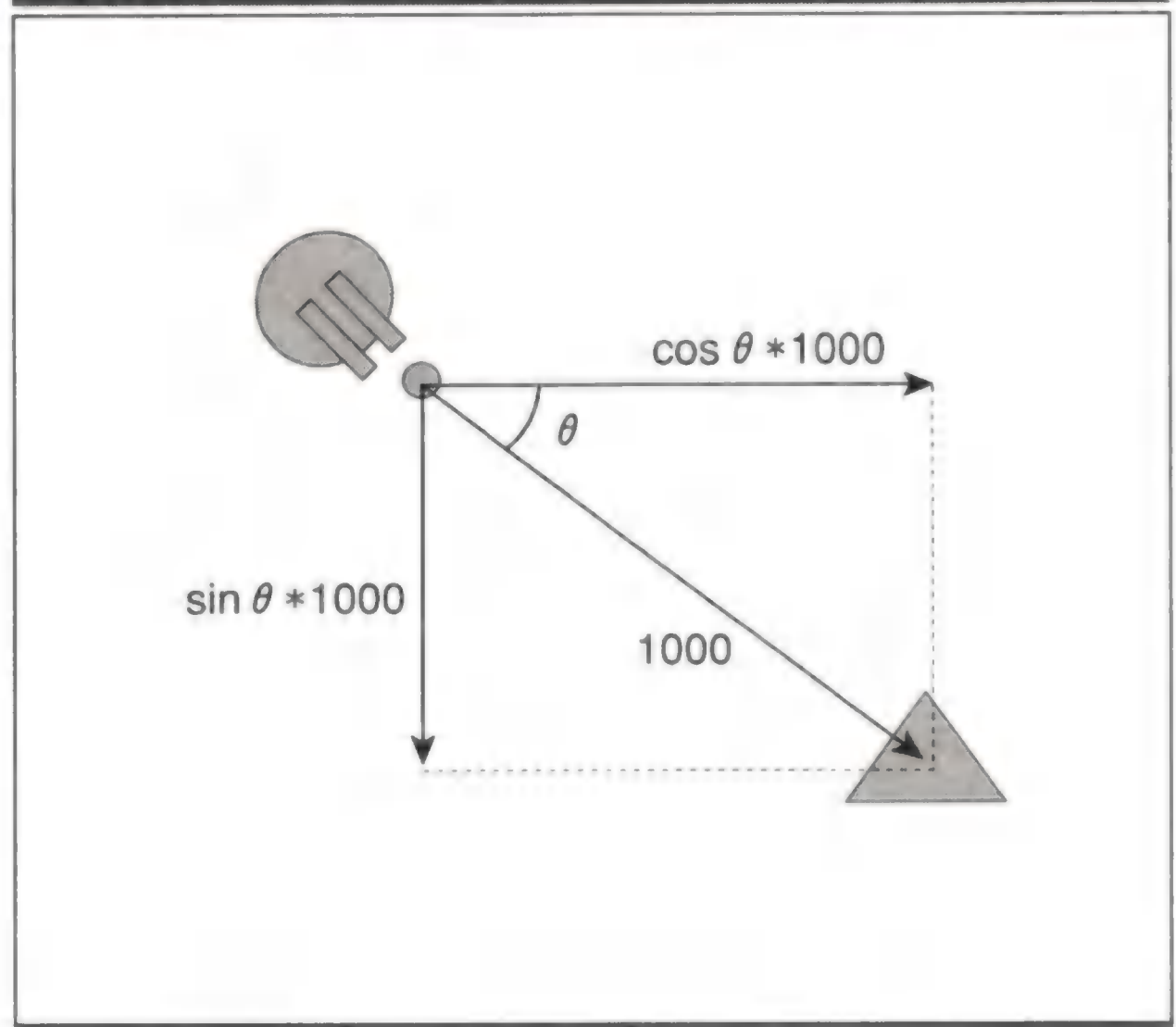



Fig. 2-18 弾を飛ばしたい方向に仮想的に自機を配置する

List 2-10 DDAを使った方向弾の初期化

```
#include <math.h>

void InitDirectedBulletDDA(
    int theta,          // 発射角度
    int ex, int ey,     // 敵の座標
    int& x, int& y,     // 弾の座標
    int& vx, int& vy,   // 弾の移動方向
    int& dx, int& dy,   // X方向とY方向の差分
```

```
    int& diff           // 誤差
) {
    // 仮想的な自機を遠くに配置する：
    // M_PIは円周率。
    int mx=cos(M_PI/180*theta)*1000;
    int my=sin(M_PI/180*theta)*1000;

    // あとはDDAを使った狙い撃ち弾の処理と同じ

    // 弾の座標を設定する
    x=ex; y=ey;

    // 弾の移動方向(vx, vy)を求める：値は1または-1
    vx=mx>ex?1:-1;
    vy=my>ey?1:-1;

    // 目標に対するX方向とY方向の差分の絶対値(dx, dy)を求める
    dx=mx>=ex?mx-ex:ex-mx;
    dy=my>=ey?my-ey:ey-my;

    // 誤差diff：dx>=dyのときはdx/2、dx<dyのときはdy/2とする
    diff=dx>=dy?dx/2:dy/2;
}
```

■ 整数型のみで処理を行う

List 2-10では実数型の関数（cosやsin）を使いますが、あらかじめ、

cos（rad）*1000
sin（rad）*1000

といった値を「0°～360°（0～2πラジアン）」に関して求め、テーブル（配列）に格納しておけば、ゲーム中の処理は完全に整数型だけですみます。テーブルを作るプログラムはList 2-11のように書きます。

List 2-12は、このテーブルmposを使って弾を発射するプログラムです。List 2-10を2-12に置き換えれば、実数型を使わなくてすむようになります。

サンプル

● 方向弾（DDA）　→ P. 312

List 2-11 位置のテーブルを作る

```
#include <math.h>

// 自機の位置を格納するテーブル
int mpos[360][2];

// テーブルを作る：
// M_PIは円周率。
void MakeTable() {
    for (int i=0; i<360; i++) {
        mpos[i][0]=cos(M_PI/180*i)*1000;
        mpos[i][1]=sin(M_PI/180*i)*1000;
    }
}
```

List 2-12 位置のテーブルを使った方向弾の初期化

```
void InitDirectedBulletDDA2(
    int theta,          // 発射角度
    int ex, int ey,     // 敵の座標
    int& x, int& y,     // 弾の座標
    int& vx, int& vy,   // 弾の移動方向
    int& dx, int& dy,   // X方向とY方向の差分
    int& diff           // 誤差
) {
    // 仮想的な自機の位置をテーブルから読み出す
    int dir=(theta%360+360)%360;
    int mx=mpos[dir][0];
    int my=mpos[dir][1];

    // あとはDDAを使って自機の方向に弾を飛ばす処理と同じ

    // 弾の座標を設定する
    x=ex; y=ey;

    // 弾の移動方向(vx, vy)を求める：値は1または-1
    vx=mx>ex?1:-1;
    vy=my>ey?1:-1;

    // 目標に対するX方向とY方向の差分の絶対値(dx, dy)を求める
    dx=mx>=ex?mx-ex:ex-mx;
    dy=my>=ey?my-ey:ey-my;

    // 誤差diff：dx>=dyのときはdx/2、dx<dyのときはdy/2とする
    diff=dx>=dy?dx/2:dy/2;
}
```

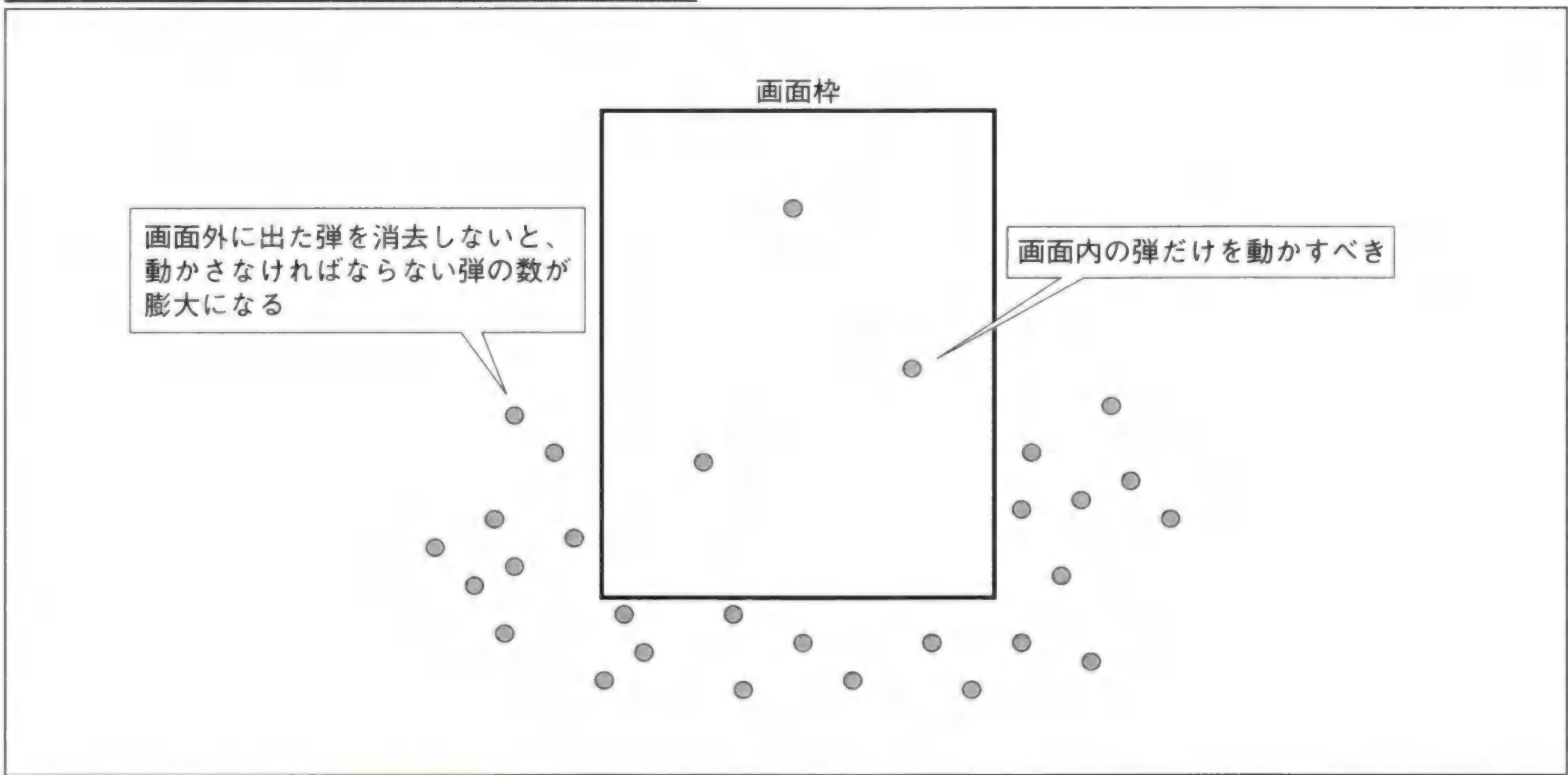
弾の消去

画面外に出た弾は消去する必要があります。画面から出た弾を残したままにしておくと、プログラムが動かさなければならない弾の数が膨大になって、処理が重くなってしまうからです(Fig. 2-19)。

Fig. 2-19 画面外に出た弾を消さない場合

画面枠

画面外に出た弾を消去しないと、動かさなければならない弾の数が膨大になる

画面内の弾だけを動かすべき

弾が画面内にあるか画面外にあるかは、弾の座標と画面枠の座標とを比べれば判定できます(Fig. 2-20)。弾の左上座標を(x0, y0)、弾の右下座標を(x1, y1)、画面枠の左上座標を(sx0, sy0)、画面枠の右下座標を(sx1, sy1)とします。このとき、弾が完全に画面外にある条件は次のように求められます。

```
(x1<=sx0 || sx1<=x0 || y1<=sy0 || sy1<=y0)
```

この条件式を使って弾が画面内にあるか画面外にあるかを判定し、画面外に出たら消去します。

なお、弾が完全に画面内にある条件は次のとおりです。

```
(sx0<=x0 && x1<=sx1 && sy0<=y0 && y1<=sy1)
```

弾の一部が画面内にある条件は次の2つのうちどちらかです。

```
!(x1<=sx0 || sx1<=x0 || y1<=sy0 || sy1<=y0)
```

```
(sx0<x1 && x0<sx1 && sy0<y1 && y0<sy1)
```

弾を消去するときに使う画面枠の座標は、実際の画面枠よりも少し広めにしておくとよいでしょう（Fig. 2-21）。画面枠ぎりぎりの領域は、画面外からやってくる弾を発生させるために使います。そのため、画面枠ぴったりで消去する弾の判定を行うと、発生させたばかりの弾を消してしまう恐れがあります。弾消去用の画面枠を広めにしておけば、発生させた弾を消してしまうことはありません。

Fig. 2-20 弾が画面外にあるかどうかの判定

(sx0, sy0)
画面枠
弾が完全に画面内にある条件：
(sx0<=x0 && x1<=sx1 && sy0<=y0 && y1<=sy)
(x0, y0)
(x1, y1)
弾の一部が画面内にある条件：
!(x1<=sx0 || sx1<=x0 || y1<=sy0 || sy1<=y0)
または
(sx0<x1 && x0<sx1 && sy0<y1 && y0<sy1)
(x0, y0)
(x1, y1)
(sx1, sy1)
弾が完全に画面外にある条件：
(x1<=sx0 || sx1<=x0 || y1<=sy0 || sy1<=y0)
(x0, y0)
(x1, y1)

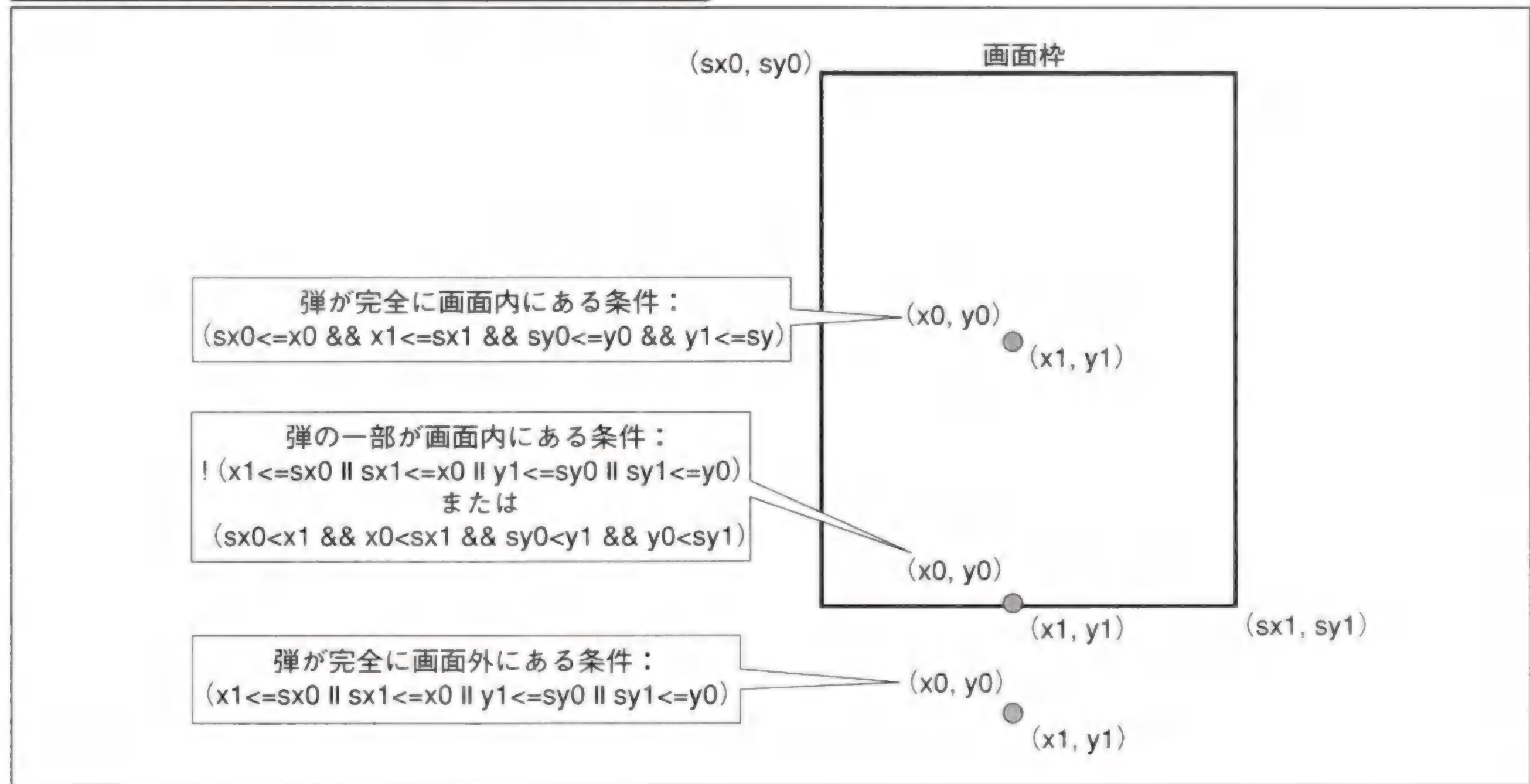
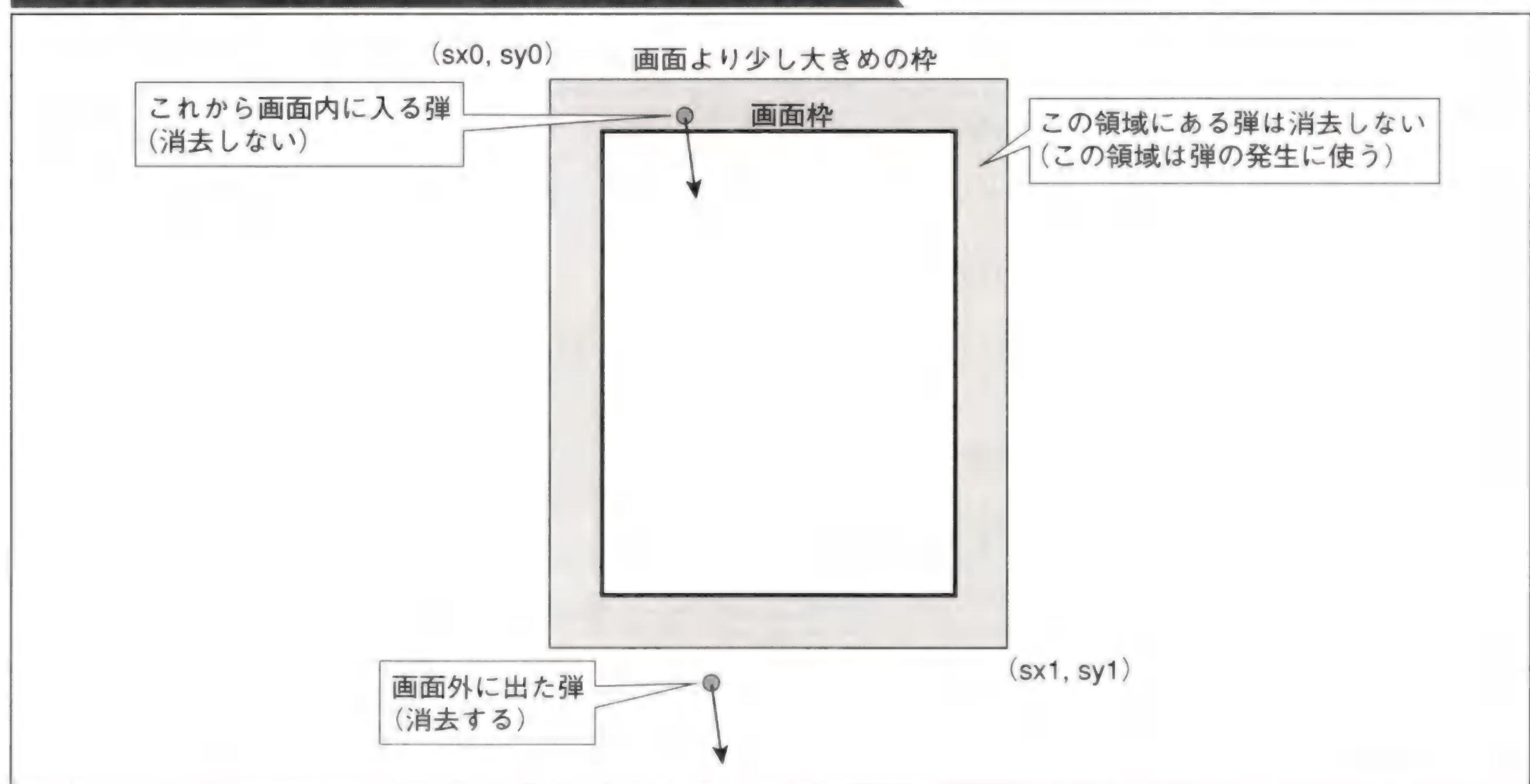



Fig. 2-21 弾を消去する枠は実際の画面枠よりも広めにする

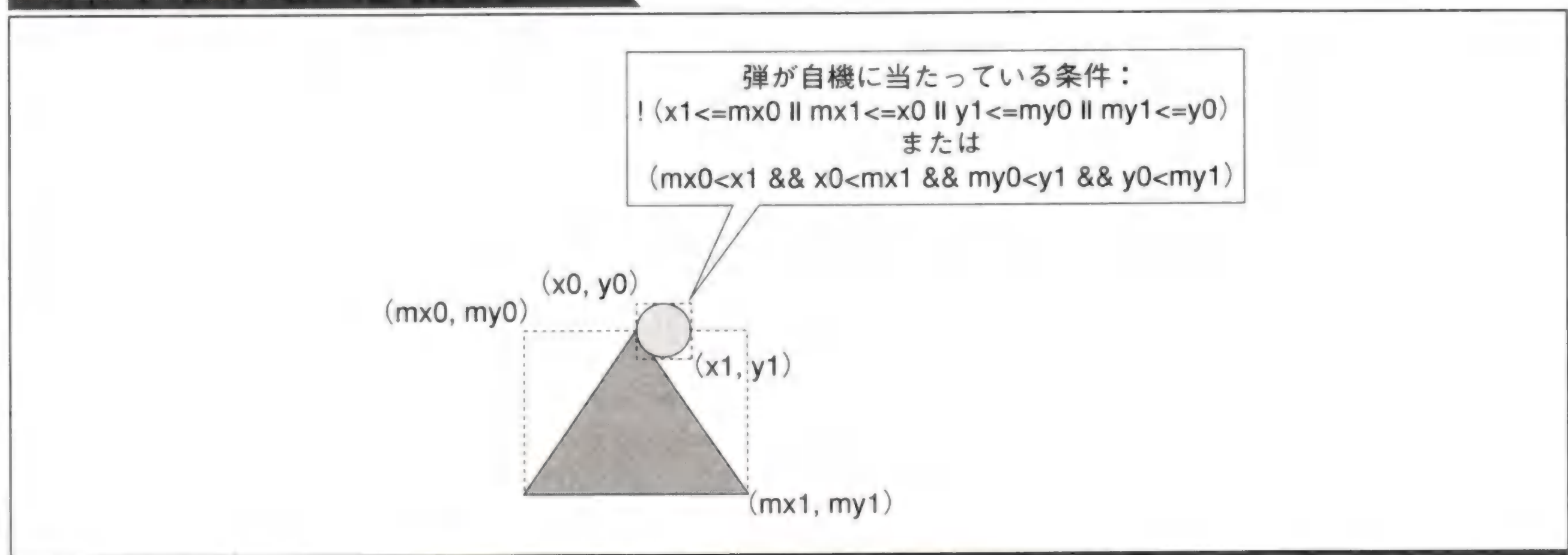
(sx0, sy0)
画面より少し大きめの枠
これから画面内に入る弾
(消去しない)
画面枠
この領域にある弾は消去しない
(この領域は弾の発生に使う)
(sx1, sy1)
画面外に出た弾
(消去する)

弾の当たり判定処理

多くのシューティングゲームでは、弾や敵が自機に命中（接触）すると自機を失うかダメージを受けるかします。このように「弾や敵が自機に当たったかどうか」を判定する処理のことを「当たり判定処理」といいます。

当たり判定処理にはさまざまな方法がありますが、オーソドックスなのは座標を使った判定方法です（Fig. 2-22）。弾の左上座標を（x0, y0）、弾の右下座標を（x1, y1）、自機の左上座標を（mx0, my0）、自機の右下座標を（mx1, my1）とします。このとき、弾が自機に当たっている条件は次の2つのうちどちらかです。

```
!(x1<=mx0 || mx1<=x0 || y1<=my0 || my1<=y0)

(mx0<x1 && x0<mx1 && my0<y1 && y0<my1)
```

実はこれは、「弾の一部が画面内にある条件」（Fig. 2-20）と同じ条件式です。弾が自機に当たるというのは、「弾の一部が自機内にある」ということだからです。

Fig. 2-22 弾と自機の当たり判定処理

判定に使われる弾や自機の座標（Fig. 2-22にて点線で示した矩形の領域）のことを、「当たり判定」と呼びます（「命中判定」や「食らい判定」と呼ぶこともあります）。「当たり判定処理」のことを「当たり判定」と呼ぶこともありますが、本書では「判定用の領域」と「判定の処理」とを区別するために、領域のことを「当たり判定」と呼び、処理のことを「当たり判定処理」と呼ぶことにします。

■ 当たり判定の設定

Fig. 2-22では当たり判定を弾や自機をすっぽり囲む外枠にしましたが、これは実際のシューティングゲームとしては大きすぎる領域です。これでは、見た目には当たっていないときにも、

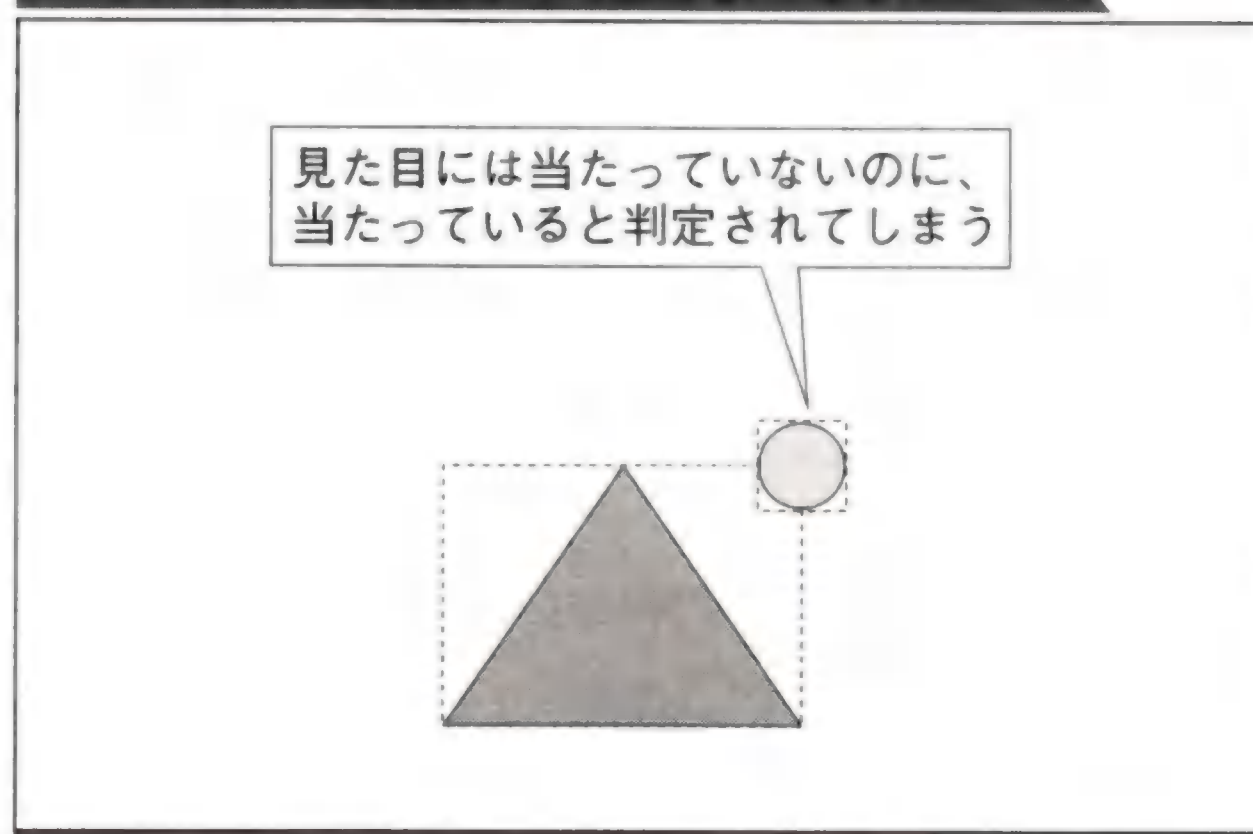
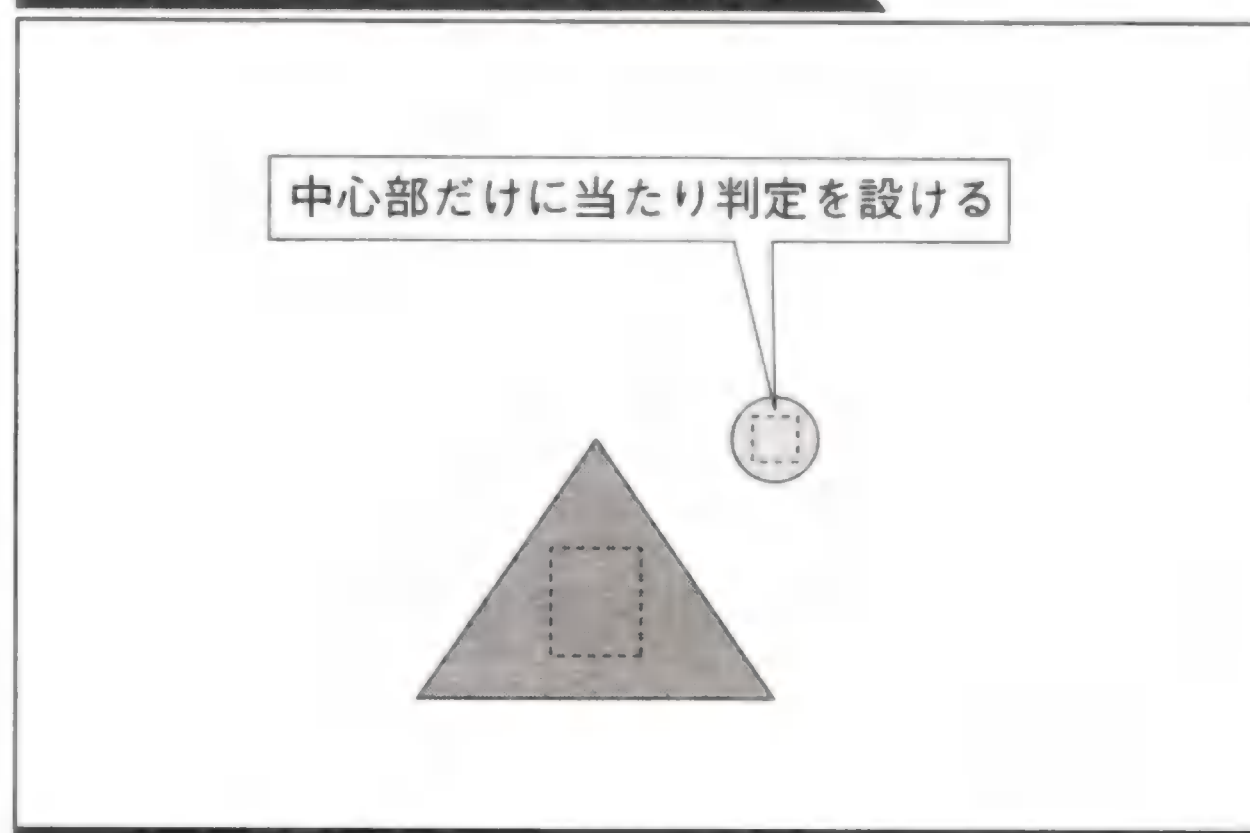
当たっていると判定されてしまうことがあります(Fig. 2-23)。ゲームを遊ぶ側の立場からすると、よけたと思ったのに弾に当たってしまうのは、非常にストレスがたまるものです。

そこで、当たり判定は弾や自機の見た目よりもやや小さめにして、弾や自機の内側だけに設けるようにします(Fig. 2-24)。これならば、「よけたつもりなのに当たってしまった」ということはありません。

Fig. 2-23 大きすぎる当たり判定の問題

見た目には当たっていないのに、
当たっていると判定されてしまう

Fig. 2-24 適度な当たり判定

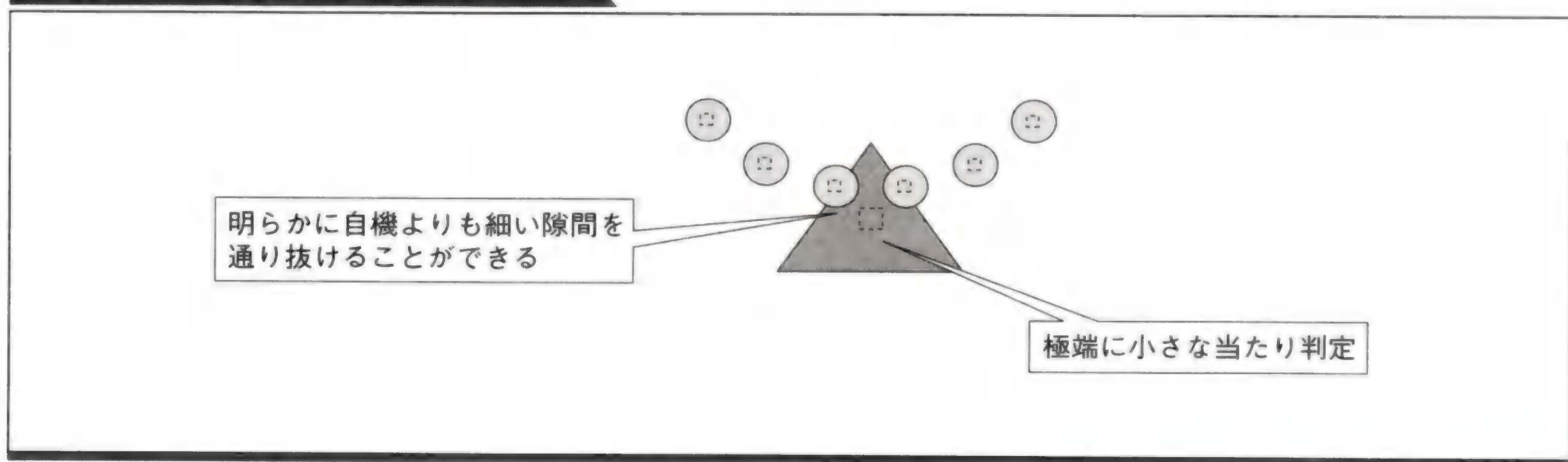
中心部だけに当たり判定を設ける

最近のシューティングゲームは当たり判定がとても小さめです。明らかに自機よりも細い弾幕の隙間でも、隙間の中央に自機を合わせれば通り抜けられてしまいます(Fig. 2-25)。極度に当たり判定を小さくして、「極度に密度の高い弾幕を抜けさせること」をゲームの目的にしている作品もあります(「サイヴァリア(→ P. 327)」など)。

当たり判定が小さいと自機はやられにくくになりますが、あまり小さすぎると「弾を避けている」という実感がわかなくなってしまいます。そのバランスをいかに調整するかが、シューティングゲーム制作者の腕の見せどころです。

たとえば「サイヴァリア」のように、自機を弾にかするせることによってレベルアップする要素を入れたり、あるいは「式神の城(→ P. 327)」のように、弾に接近することによってスコアの増加率が上がる要素を入れたりするのも、当たり判定を小さくしつつ緊張感のあるゲーム性を演出する方法でしょう。

Fig. 2-25 極度に小さな当たり判定

明らかに自機よりも細い隙間を
通り抜けることができる

極端に小さな当たり判定

n-way弾

n-way弾というのは、扇形に飛ぶ弾のことです（Fig. 2-26）。弾の数は3方向や5方向などがありますが、ゲームによっては画面をおおいつくすほど多くの方向に弾が飛ぶこともあります。

n-way弾は「方向弾（→ P. 23）」の応用で作ることができます。中心を飛ぶ弾のわきに、適当な角度で軸線をずらしていくつかの弾を飛ばせばよいのです（Fig. 2-27）。たとえば、角度θだけずらして中心線の左右に2個ずつ弾を飛ばせば、5-way弾になります。

中心線には弾を飛ばさないn-way弾にする場合もあります（Fig. 2-28）。この場合は中心線からθ/2だけずらして左右に弾を飛ばし、あとはθずつずらして弾を飛ばします。

Fig. 2-26 n-way弾

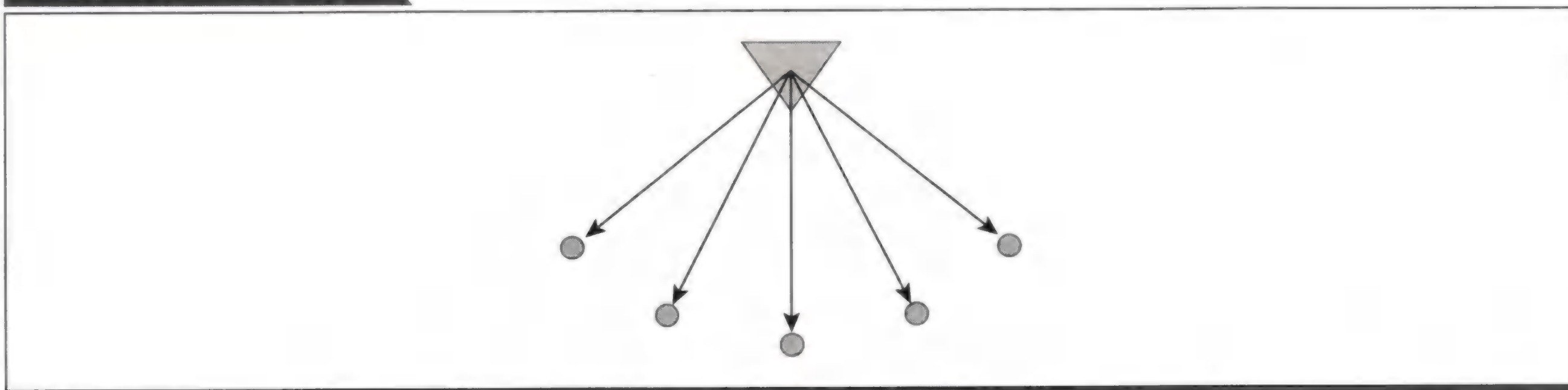



Fig. 2-27 n-way弾の作成方法

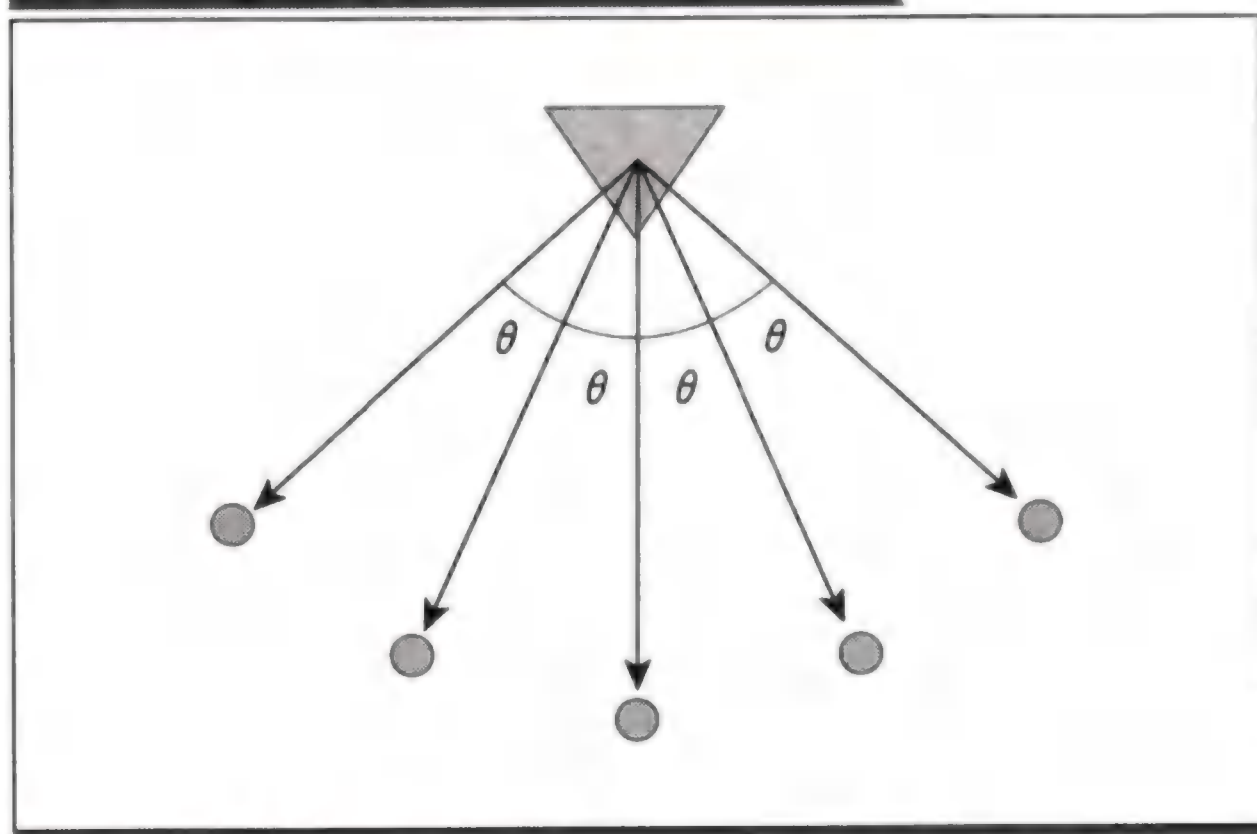



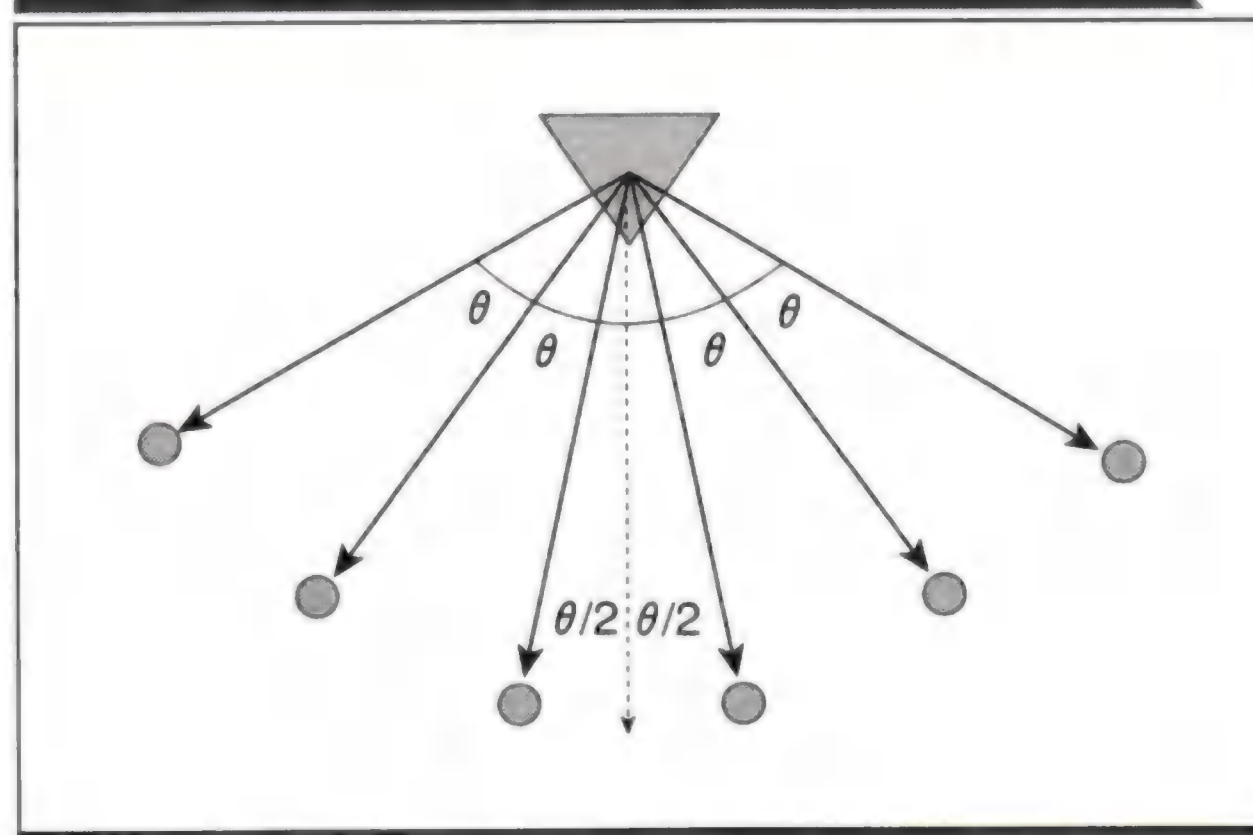
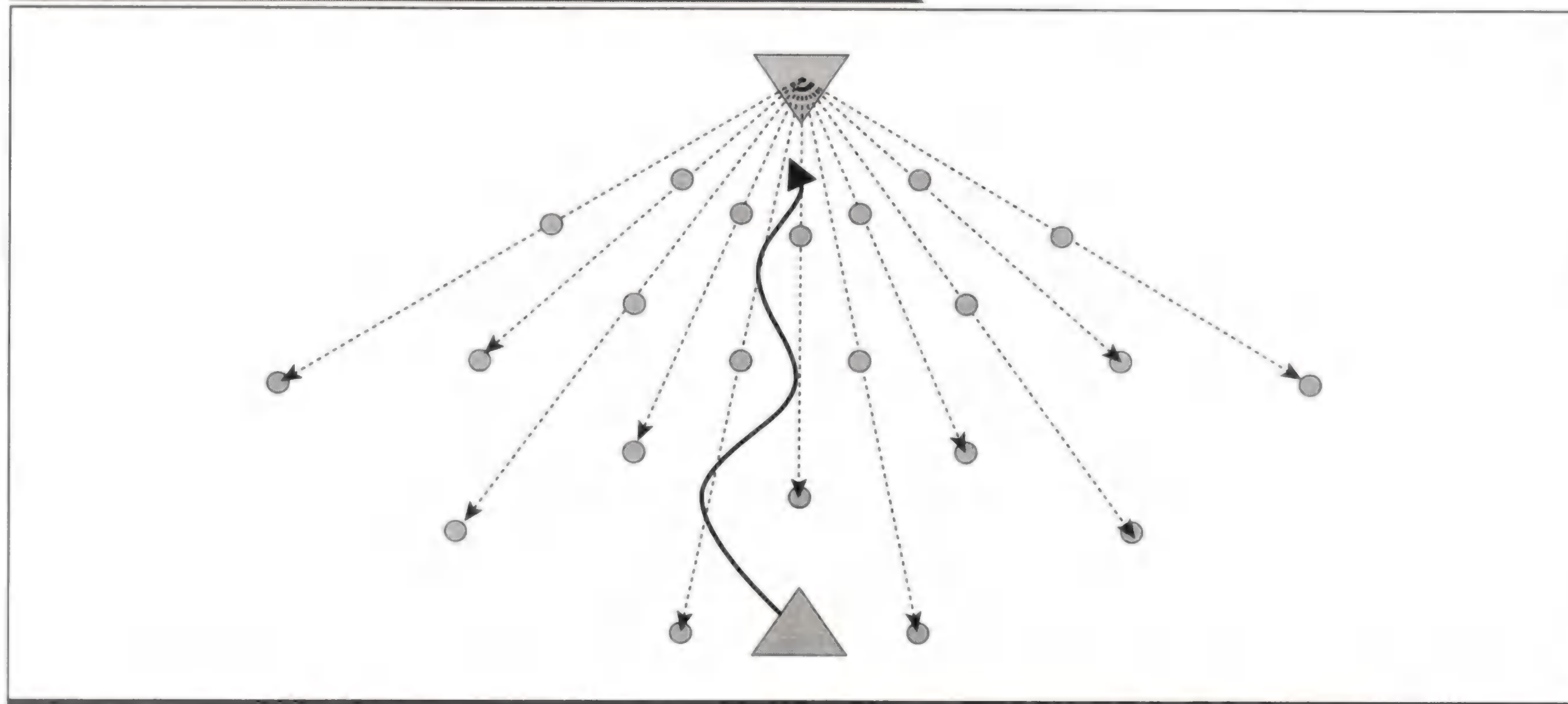
Fig. 2-28 中心線に弾を飛ばさないn-way弾

ここでは便宜上、Fig. 2-27のn-way弾を「奇数パターン」と呼び、Fig. 2-28のn-way弾を「偶数パターン」と呼ぶことにします。多くのシューティングゲームでは、奇数パターンと偶数パターンを状況に応じて使い分けます。敵が正面方向に弾を撃つ場合、偶数パターンは敵の正面で回避できますが、奇数パターンは敵の正面から左右に少しずれたところで回避する必要があります。そこで、特にステージ最後などに出てくるボスキャラは奇数パターンと偶数パターンを交互に撃つことがよくあります。こうすると回避ポイントが絶え間なく動くので、1つのパターンを繰り返すよりもゲーム性が高まるのです（Fig. 2-29）。

Fig. 2-29 2種類のn-way弾が交互にくる場合の回避コース

n-way弾の発射

n-way弾を発射するには、まず中心となる弾の速度を決めてから、その速度ベクトルを中心線の左右に回転させたものをほかの弾の速度とします。中心となる弾の速度を(vx0, vy0)とすると、これをθだけ回転させた速度ベクトル(vx, vy)はList 2-13のようなプログラムで求められます。

List 2-13 速度ベクトルを回転させる

```
#include <math.h>

void RotateVelocity(
    float theta,            // 回転角度
    float vx0, float vy0,   // 元の速度
    float& vx, float& vy    // 回転後の速度
) {
    // thetaをラジアンに変換し、cosとsinを求める：
    // M_PIは円周率。
    float rad=M_PI/180*theta;
    float c=cos(rad), s=sin(rad);

    // 速度ベクトル(vx0, vy0)を回転させた(vx, vy)を求める
    vx=vx0*c-vy0*s;
    vy=vx0*s+vy0*c;
}
```

List 2-13の要領で中心線からθ、θ*2、$-\theta$、$-\theta$*2だけ回転させた弾を発射すれば、奇数パターンの5-way弾を発射することができます(Fig. 2-30)。また、θ/2、θ*3/2、θ*5/2、$-\theta$/2、$-\theta$*3/2、$-\theta$*5/2だけ回転させた弾を発射すると、偶数パターンの6-way弾になります(Fig. 2-31)。

n-way弾の発射処理をまとめたものがList 2-14です。中心となる弾の速度、弾と弾との間の角度、そして弾の数を基に、n-way弾を構成するn個の弾の速度を計算します。

n-way弾で中心線となる弾を狙い撃ち弾にすると、「自機に向かってくるn-way弾」を作ることができます。実際のゲームでは、この「自機に向かってくるn-way弾」と「固定方向に発射されるn-way弾」の両方を使い分けることが多いようです。あるいは「自機が敵の左側にいるときにはやや左方向にn-way弾を撃ち、右側にいるときにはやや右方向に撃つ」といった性質のn-way弾も、ボスキャラなどで見かけることがあります。

サンプル

n-way弾 → P. 312

Fig. 2-30 5-way弾の発射方法

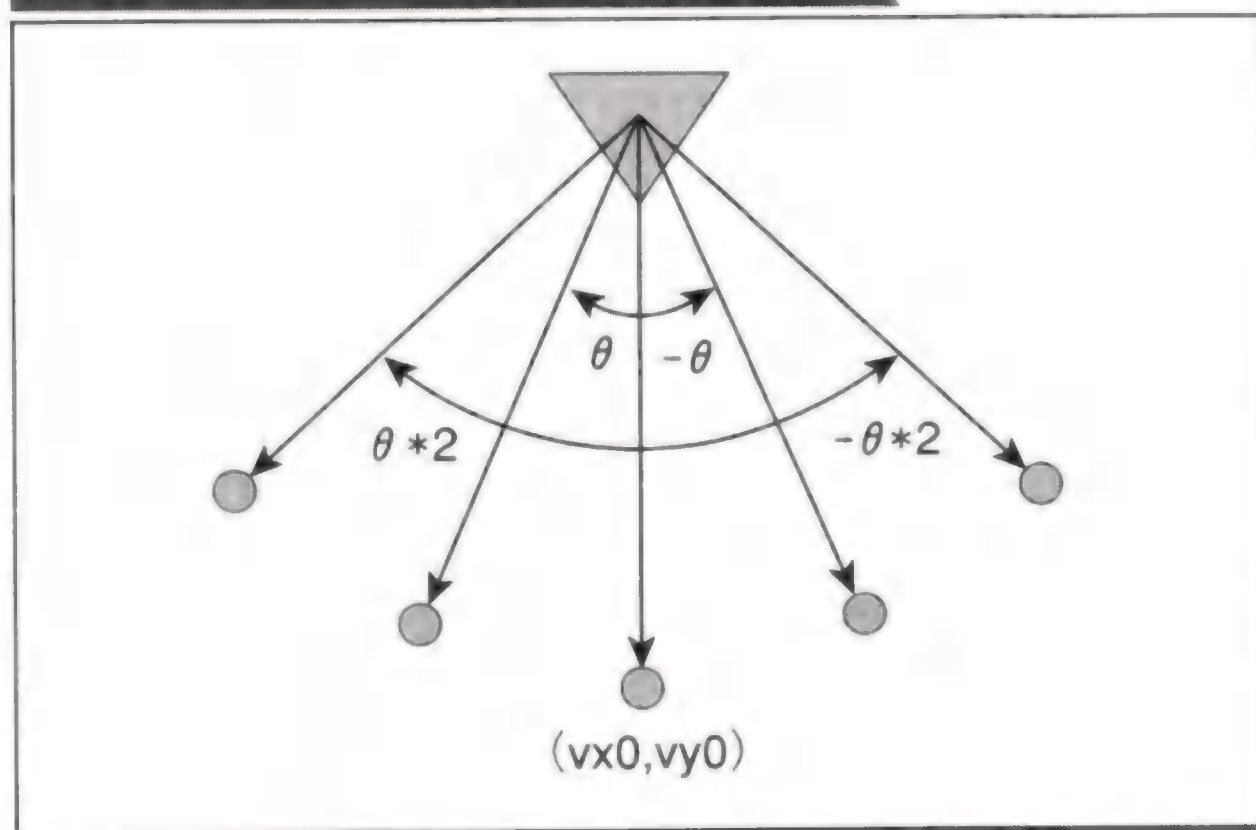
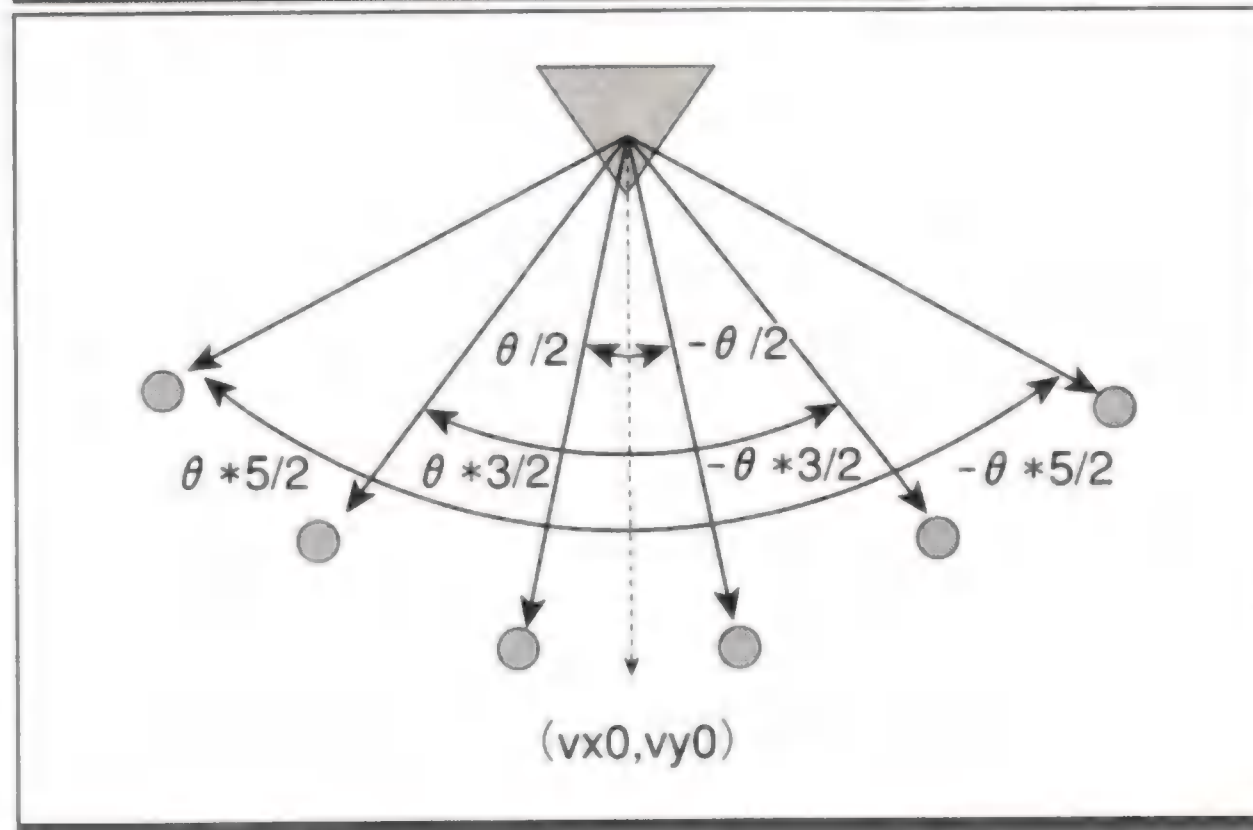



Fig. 2-31 6-way弾の発射方法

List 2-14 n-way弾の初期化

```
#include <math.h>

void InitNWayBullets(
    float vx0, float vy0,    // 中心となる弾の速度
    float theta,             // 弾と弾の間の角度
    int n,                   // 弾の数
    float vx[], float vy[]   // n-way弾の速度
) {
    // 弾と弾の間の角度をラジアンに変換する
    float rad_step=M_PI/180*theta;

    // 端の弾と中心の弾との間の角度を計算する
```

```
    float rad=n%2 ? -n/2*rad_step : (-n/2+0.5)*rad_step;

    // n個の弾の速度を計算する
    for (int i=0; i<n; i++, rad+=rad_step) {

        // (vx[i], vy[i])を求める：
        // 速度ベクトル(vx0, vy0)をradだけ回転させる。
        float c=cos(rad), s=sin(rad);
        vx[i]=vx0*c-vy0*s;
        vy[i]=vx0*s+vy0*c;
    }
}
```

円形弾

ボスキャラなどは、360度全方位に向かってほぼ同時に弾を撃つことがあります（Fig. 2-32）。このように円形に広がる弾は、方向弾（→ P. 23）の応用で作ることができます。弾の射出方向を一定角度ずつずらしながら全方位に弾を飛ばせばよいのです。Fig. 2-32の場合には全部で16個の弾があるので、22.5°（=360/16）ずつ角度をずらしながら弾を発射します。

n-way弾と同様に、円形弾にも「奇数パターン」と「偶数パターン」があります（→ P. 33）。Fig. 2-33はFig. 2-32のもう1つのパターンで、Fig. 2-32の角度を11.25°（=360/16/2）だけずらしたものです。多くのゲームではFig. 2-32と2-33のように角度がずれたパターンを交互に使ってゲーム性を高めています。

Fig. 2-32 円形弾

22.5°

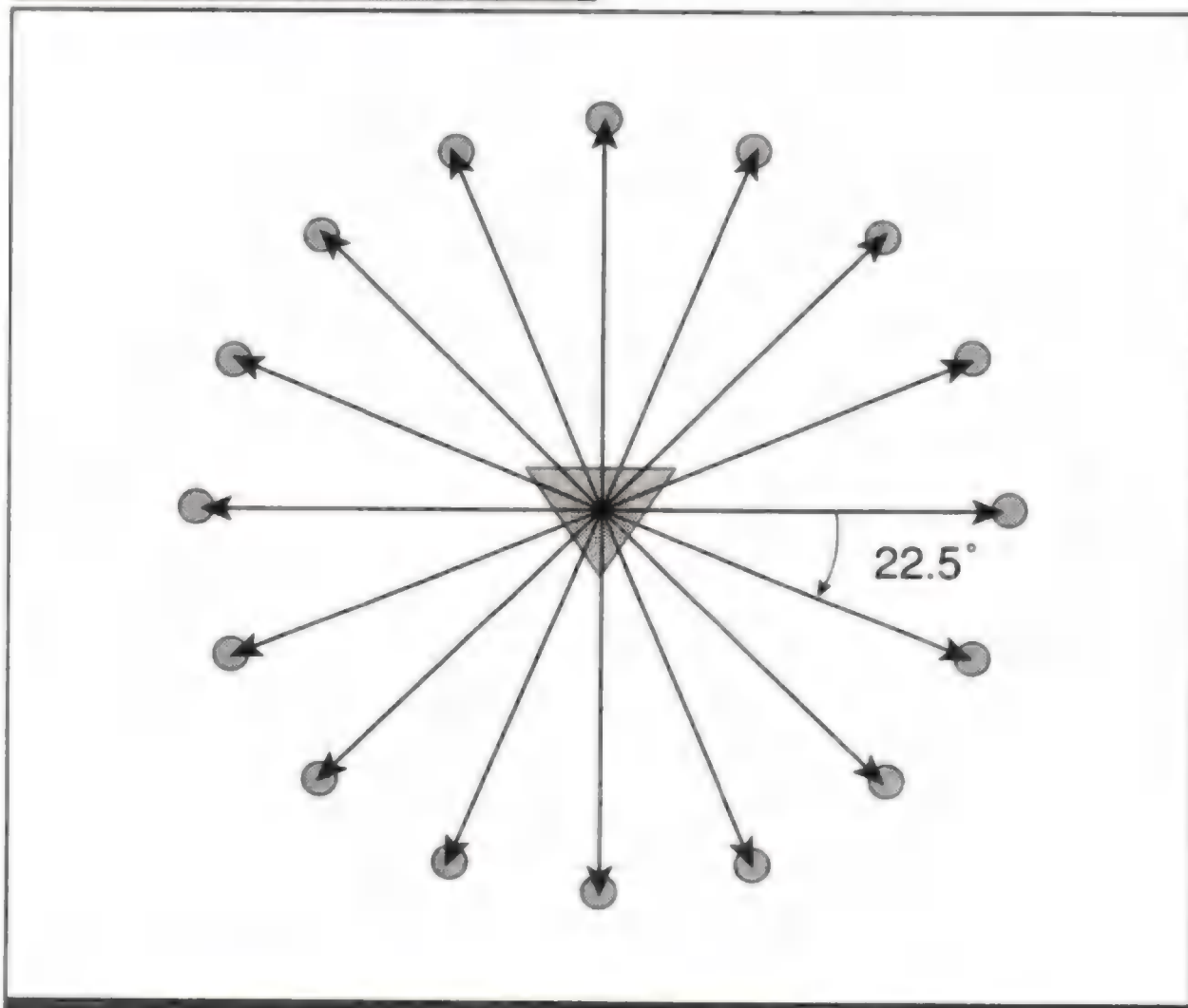
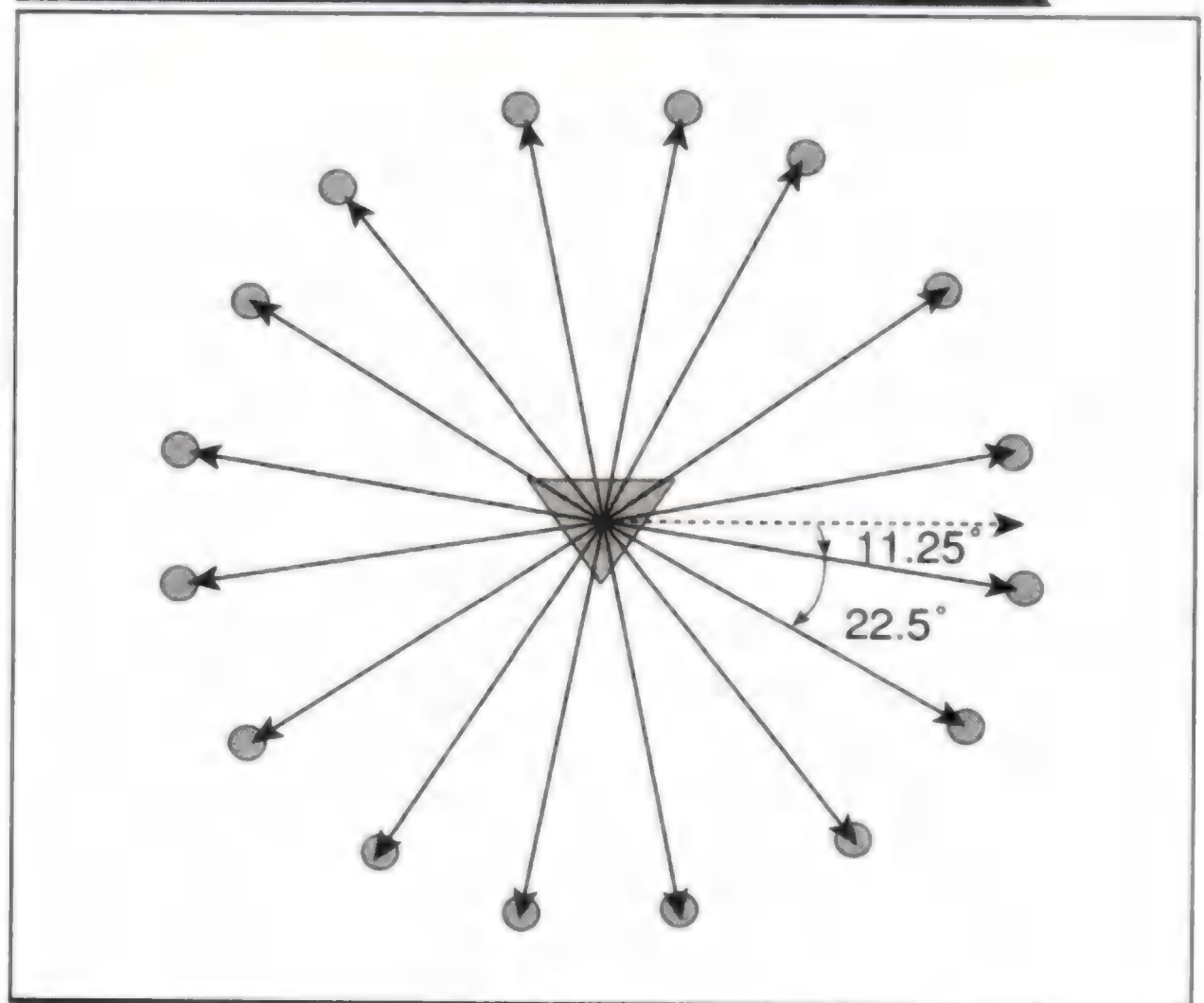



Fig. 2-33 円形弾のもう1つのパターン

11.25°
22.5°

List 2-15は円形弾を発射するプログラムです。このプログラムでは引数oddをtrueにすると奇数パターンになり、falseにすると偶数パターンになります。

円形弾を使うときには、奇数パターンと偶数パターンを適宜撃ち分けると面白くなるでしょう。さらに狙い撃ち弾などを少し混ぜると、弾幕が不規則になるので、ぐっと難易度を上げることができます。

サンプル

円形弾 → P. 313

List 2-15 円形弾の初期化

```
#include <math.h>

void InitCircleBullets(
    int n,                       // 弾の数
    float speed,                 // 弾の速さ
    bool odd,                    // 奇数パターンのときtrue
    float vx[], float vy[]       // 円形弾の速度
) {
    // 弾と弾との間の角度を計算する
    float rad_step=M_PI*2/n;

    // 最初の弾の角度を計算する：
    // 奇数パターンのときにはrad_step/2だけずらす
    float rad=odd ? rad_step/2 : 0;

    // n個の弾の速度を決める：
    // 速さspeedで角度radの方向に飛ぶ弾の速度を求める。
    // これは方向弾を飛ばす処理の応用。
    for (int i=0; i<16; i++, rad+=rad_step) {
        vx[i]=cos(rad)*speed;
        vy[i]=sin(rad)*speed;
    }
}
```

分裂弾

発射されたあと、いくつもに分かれる弾です（Fig. 2-34）。最初に遅くて大きめの弾が発射され、それがあとから速い小さめの弾に分裂するパターンなどをよく見かけます。

この分裂弾は、狙い撃ち弾（→ P. 10）とn-way弾（→ P. 33）の組み合わせで作ることができます。方向弾（→ P. 23）とn-way弾の組み合わせでもOKです。どちらの場合も、分裂弾のアルゴリズムはFig. 2-35のようになります。

Fig. 2-34 分裂弾

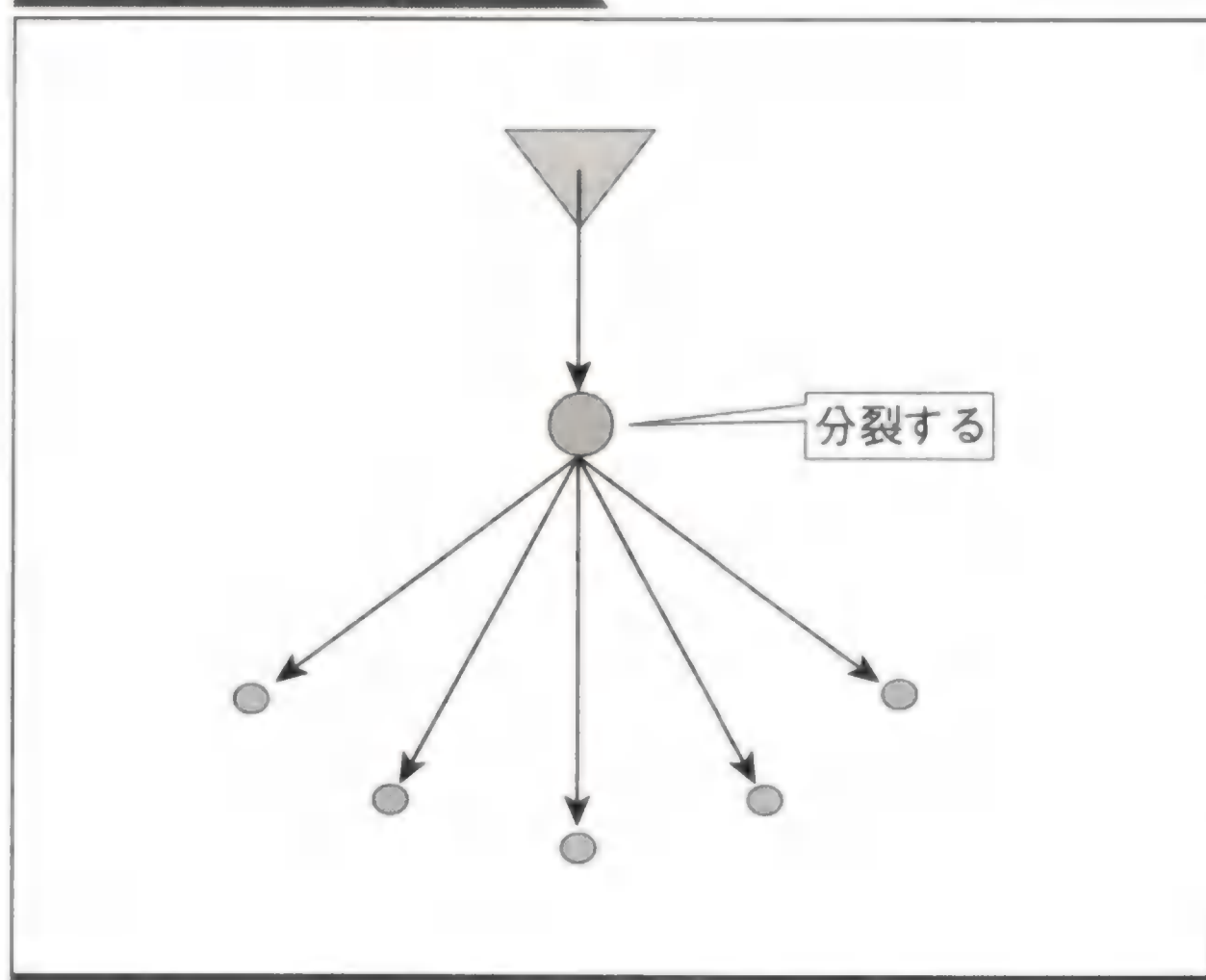
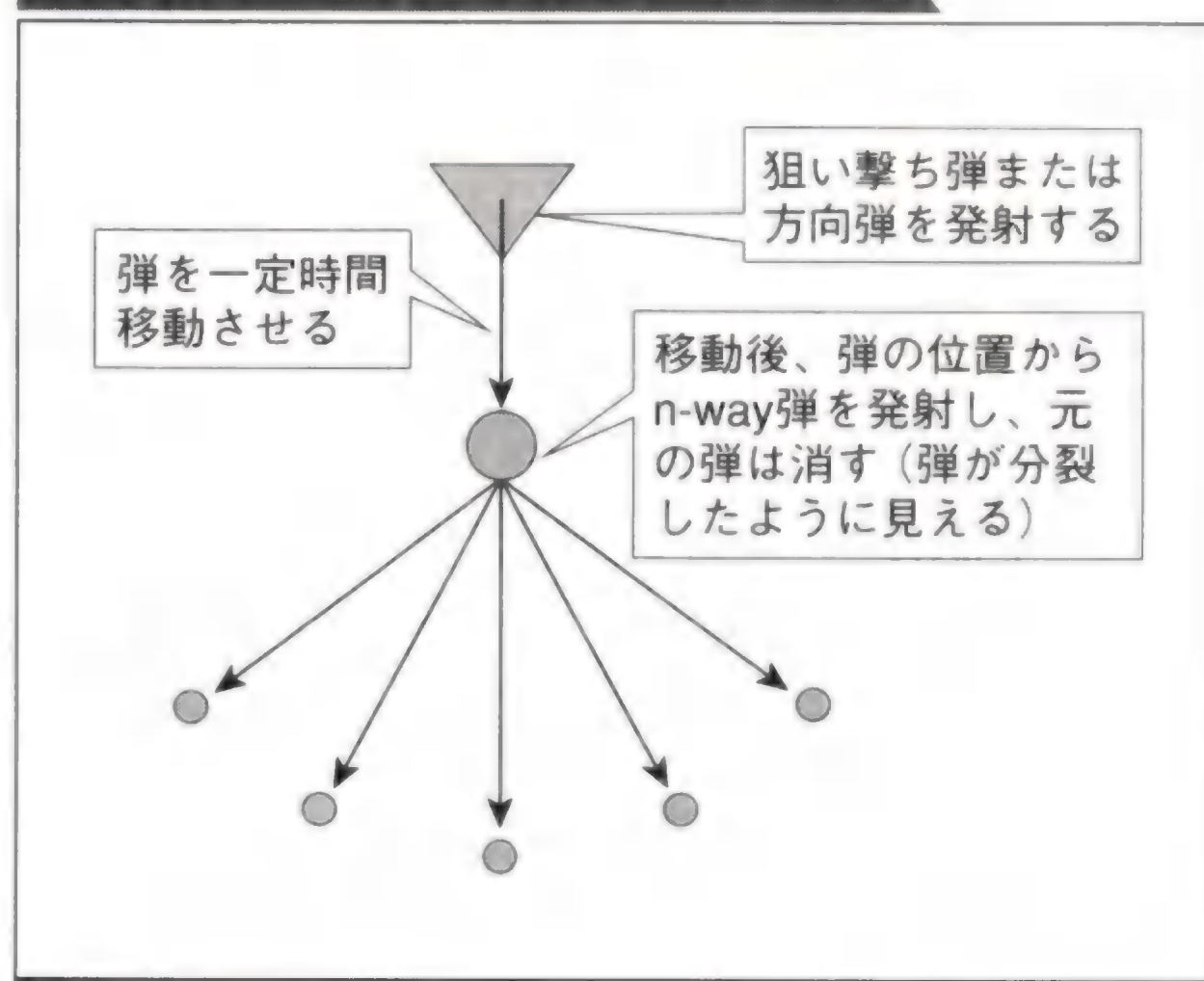



Fig. 2-35 分裂弾のアルゴリズム

まず1個の弾を発射し、一定時間移動させます。移動のあと、その1個の弾を消して、弾の位置からn-way弾を発射します。結果として弾は分裂したように見えます。この方法を応用すれば、分裂した弾が再度分裂したり、分裂後の弾が自機を追尾したりといった特殊な弾を作ることもできます。

分裂弾を使うときには、「いかにも分裂しそう」な雰囲気の演出を忘れてはいけません。「分裂しそうに見えない弾」が分裂しても、プレイヤーにとっては負担になるだけです。プレイヤーが弾の分裂を予期して準備ができるように、弾の大きさや動きや色などを通常の弾とは変える必要があります。

たとえば弾を大きめにして、動きを遅くして点滅するようにすれば、プレイヤーに対して「この弾は普通の弾とは違いますよ」というメッセージを伝えることができるでしょう。さらに分裂や爆発のアニメーションを加えれば、見た目にも楽しくなります。

サンプル

分裂弾　→ P.313

誘導弾

自機を追尾する弾です（Fig. 2-36）。自機の方向に向かって直進する弾とは違い、この弾は発射されたあとにも向きを変えながら自機を追いかけます。自機が動いたときにも追いかけてくるので、直進するだけの弾よりも回避が難しくなります。

誘導弾を作るには、弾の移動中に進行方向を自機の位置に合わせて修正します（Fig. 2-37）。弾を移動させるたびに、弾が自機の新しい位置に向かうように方向を変更すればよいのです。新しい方向は、自機の新しい位置に向かって弾を撃つときと同じ方法で計算することができます。

List 2-16はFig. 2-37の方法で誘導弾を動かすプログラムです。狙い撃ち弾（→ P. 10）の初期化処理と移動処理を連結したようなプログラムになります。弾を移動させるたびに、弾の速度を再計算するわけです。

Fig. 2-36 誘導弾

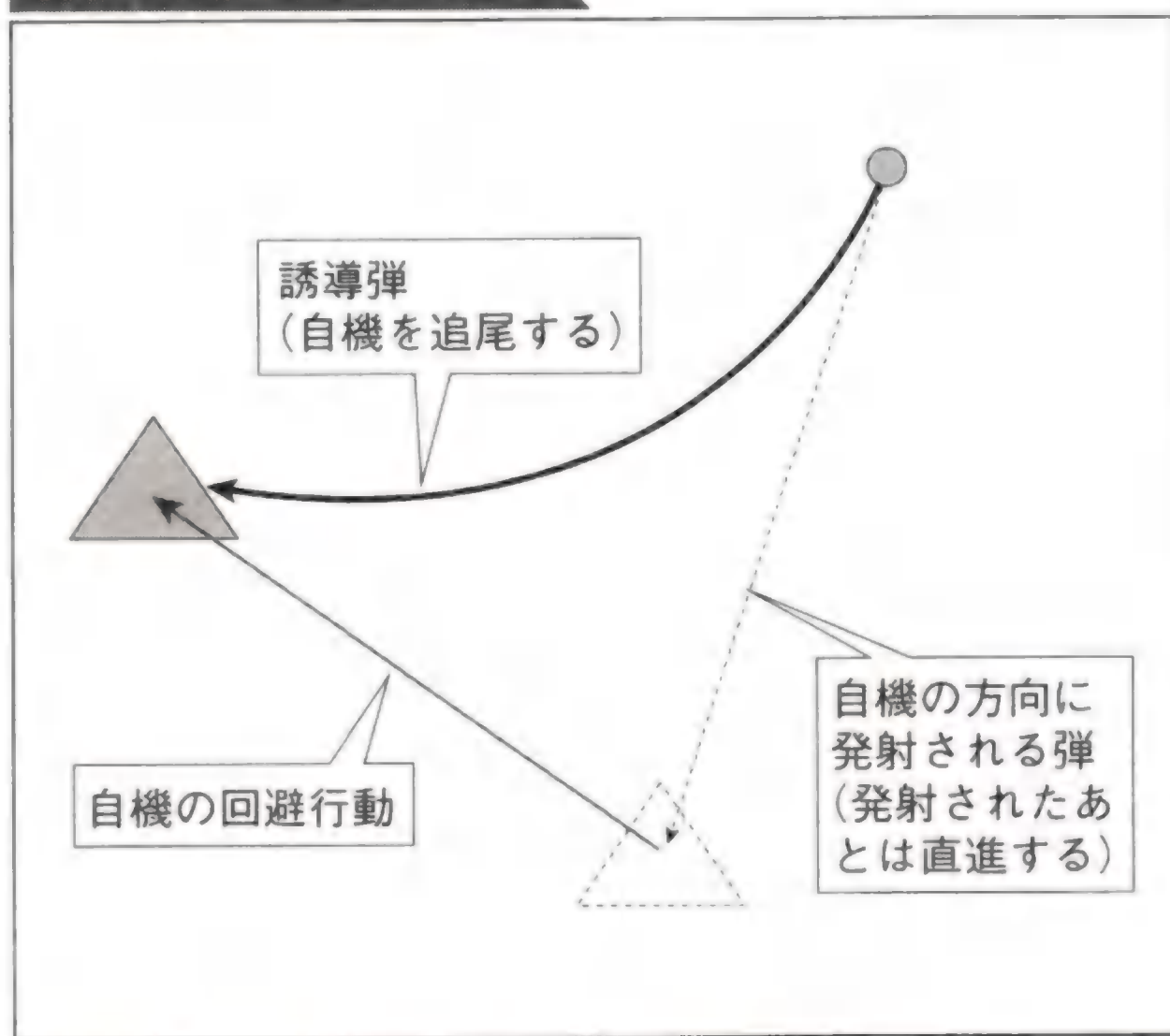
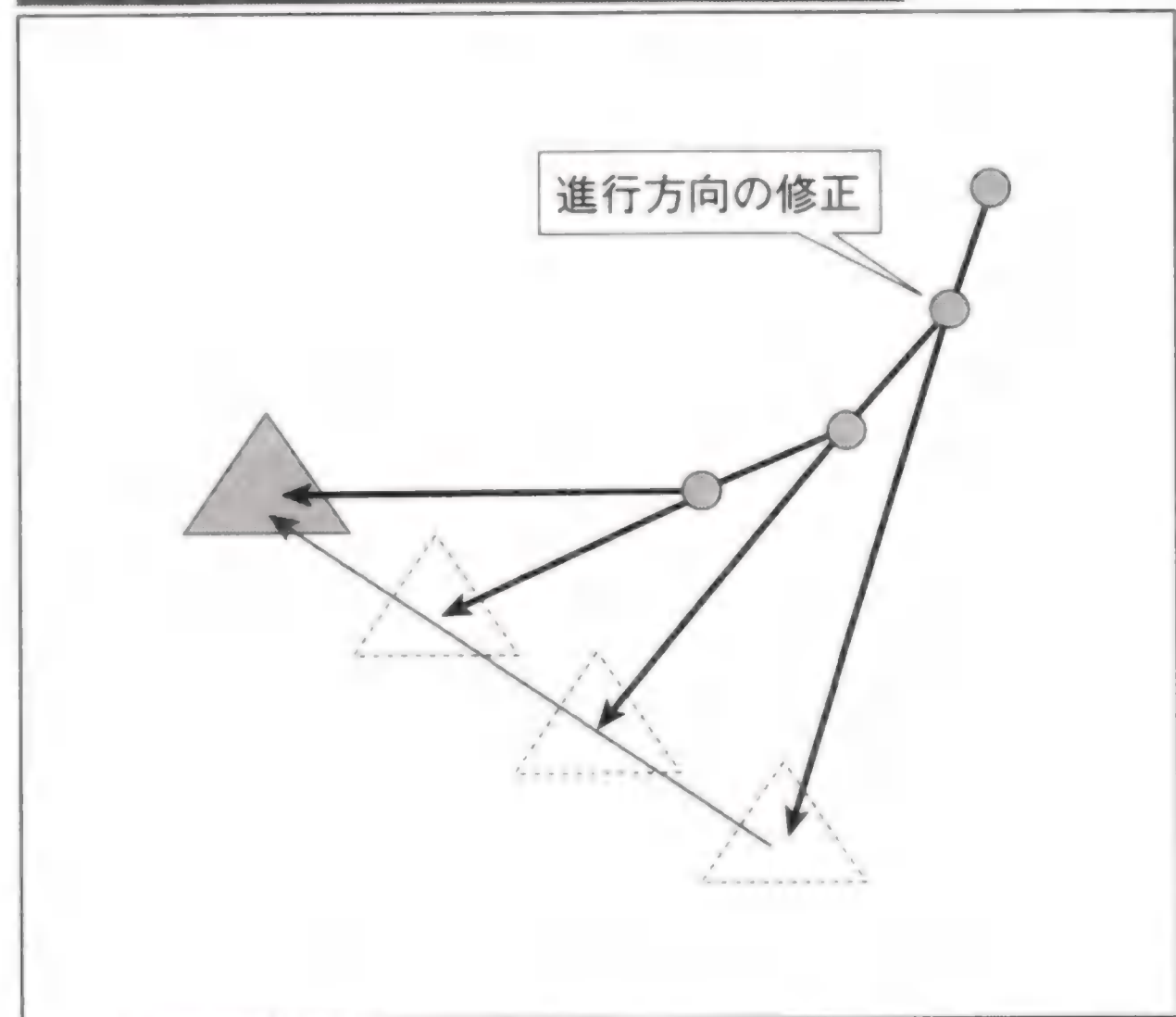



Fig. 2-37 誘導弾のアルゴリズム

List 2-16 誘導弾の移動

```
#include <math.h>

void MoveSimpleHomingBullet(
    float& x, float& y,   // 弾の座標
    float mx, float my,   // 自機の座標
    float speed           // 弾の速さ
) {
    // 目標までの距離dを求める
    float d=sqrt((mx-x)*(mx-x)+(my-y)*(my-y));
```

```
    // 弾の速度(vx, vy)を求める：
    // 速さが一定値speedになるようにする。
    // 目標までの距離dが0のときには速度を下方向にする。
    float vx, vy;
    if (d) {
        vx=(mx-x)/d*speed;
        vy=(my-y)/d*speed;
    } else {
        vx=0;
        vy=speed;
    }

    // 弾の座標(x, y)を更新して、弾を移動させる
    x+=vx;
    y+=vy;
}
```

■ 弾の誘導をあまくする

List 2-16の方法で誘導弾は作れますが、実はこの方法には欠点があります。この誘導弾は自機を無限に追い続けてしまうのです。それでは、自機がどんな回避行動をとっても、弾は正確に自機の新しい位置を目指すので、いつかは必ず命中します。これでは弾を回避することができないのでゲームとして成立しません。

弾の誘導を少しあまくしたほうが、ゲームとしては面白くなります。そこでFig. 2-38のように、弾の旋回角度に上限を設けます。弾は自機に向かって旋回しますが、一度の旋回では決められた角度を超えて曲がらないものとします。これなら自機がたくみに動けば弾をよけることができるので、適度なゲーム性が生まれます。

「旋回角度に上限を設ける」ということをもう少し厳密に考えたのがFig. 2-39と2-40です。Fig. 2-39のように自機の方向が弾の旋回可能範囲の外にあるときには、弾が旋回角度の上限で曲がるようにします。一方、Fig. 2-40のように自機の方向が弾の旋回可能範囲内にあるときには、弾を自機の方向に曲がらせます。

このように、弾の動きとしては、旋回角度の上限で曲がる場合と、自機の方向に曲がる場合とがあります。これに旋回角度の上限で反対方向に曲がる場合（逆回りで追跡する）を加えると、全部で3通りの旋回方向があることになります（Fig. 2-41）。

Fig. 2-38 旋回角度に上限を設ける

旋回角度に
上限を設ける

Fig. 2-39 自機の方向が旋回可能範囲外の場合

自機の方向が旋回可能
範囲外なので、旋回角
度の上限で曲がる

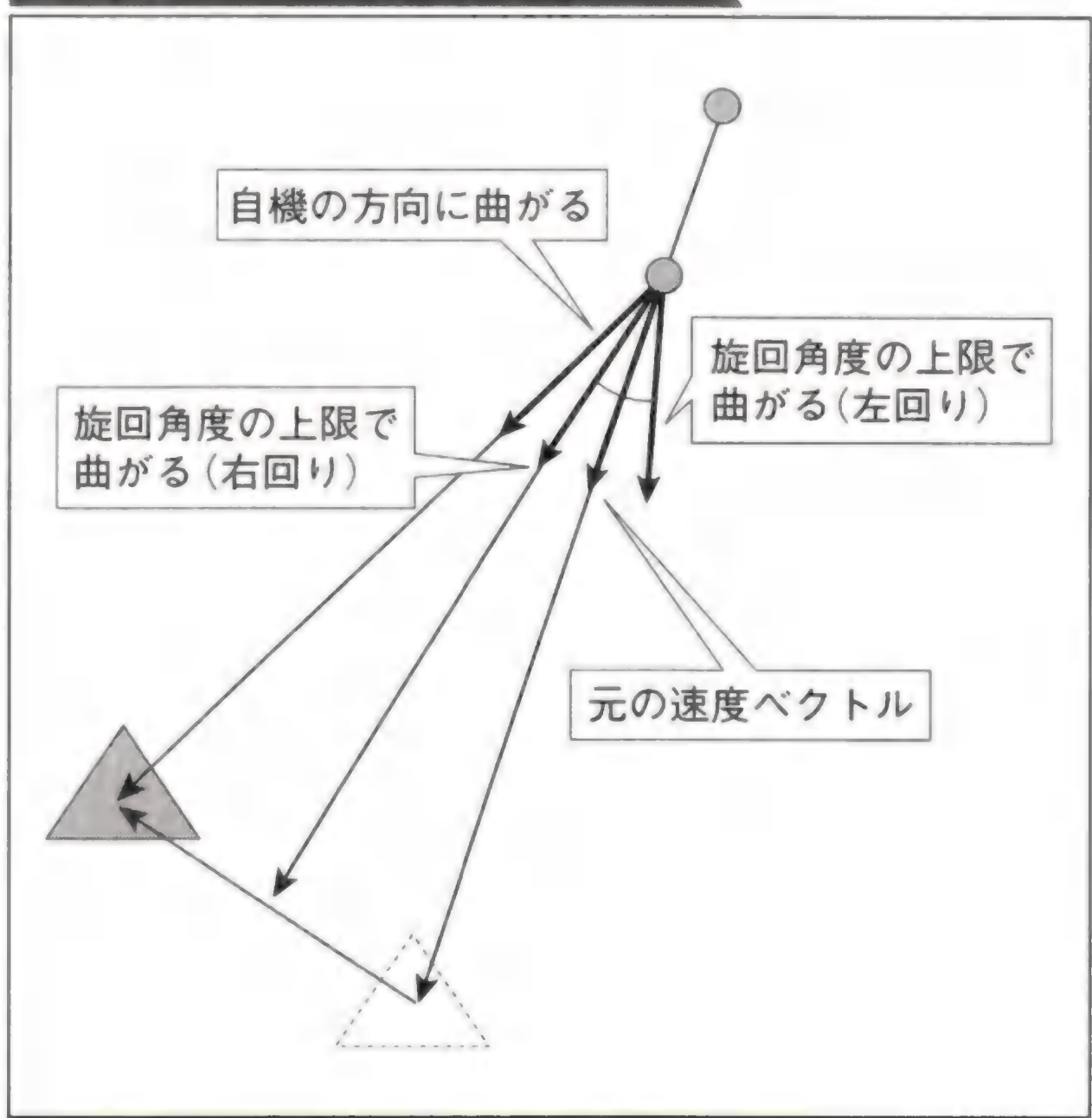
Fig. 2-40 自機の方向が旋回可能範囲内の場合

自機の方向が旋回可能
範囲内なので、自機の
方向へ曲がる

Fig. 2-41 3通りの旋回方向

自機の方向に曲がる

旋回角度の上限で
曲がる（左回り）

旋回角度の上限で
曲がる（右回り）

元の速度ベクトル

■ 旋回方向の選択

問題は、3通りのうちのどの旋回方向を選ぶかです。まず、自機の方向が旋回可能範囲内にあるかどうかで、自機の方向に曲がるか旋回角度の上限で曲がるかが決まります。「角度を比べるとなると、アークタンジェントなどの三角関数を使わなくてはならないのでは!?」と一瞬青ざめますが、実はベクトルの内積を使うだけですみます。

弾の元の速度ベクトルをv0、自機のほうに向かう弾の速度ベクトルをv1、旋回角度の上限に合わせた速度ベクトルをv2とします。そして、v0・v1という内積とv0・v2という内積の大小を比べます。v0・v1のほうが小さいときには、自機の方向は旋回可能範囲外です。逆に大きいときには旋回可能範囲内になります。

その理由はFig. 2-42と2-43からわかります。ここでは、v2の逆回りに相当するベクトルをv3としました。Fig. 2-42のようにv1がv2とv3の外側にあるときには、v1からv0への射影はv2からv0への射影よりも短くなります。つまり次のとおりです。

$$|v1|\cos\phi < |v2|\cos\theta$$

ここで2つの内積v0・v1とv0・v2はそれぞれ、

$$v0 \cdot v1 = |v0||v1|\cos\phi$$
$$v0 \cdot v2 = |v0||v2|\cos\theta$$

なので、

$$|v1|\cos\phi < |v2|\cos\theta$$

と、

$$|v0||v1|\cos\phi < |v0||v2|\cos\theta$$

つまり、

$$v0 \cdot v1 < v0 \cdot v2$$

とは同じことです。したがって、v0・v1がv0・v2よりも小さいときには、自機の方向は旋回可能範囲外になります。同じようにFig. 2-43に関しても、v1がv2とv3の内側にあるときには、v1からv0への射影はv2からv0への射影よりも長くなります。つまり次のとおりです。

$$|v1|\cos\phi > |v2|\cos\theta$$

これは、

$$|v0||v1|\cos\phi > |v0||v2|\cos\theta$$

つまり、

$$v0 \cdot v1 > v0 \cdot v2$$

と同じことです。したがって、v0・v1がv0・v2よりも大きいときには、自機の方向は旋回可能範囲内になります。

Fig. 2-42 旋回可能範囲外の場合には内積が小さくなる

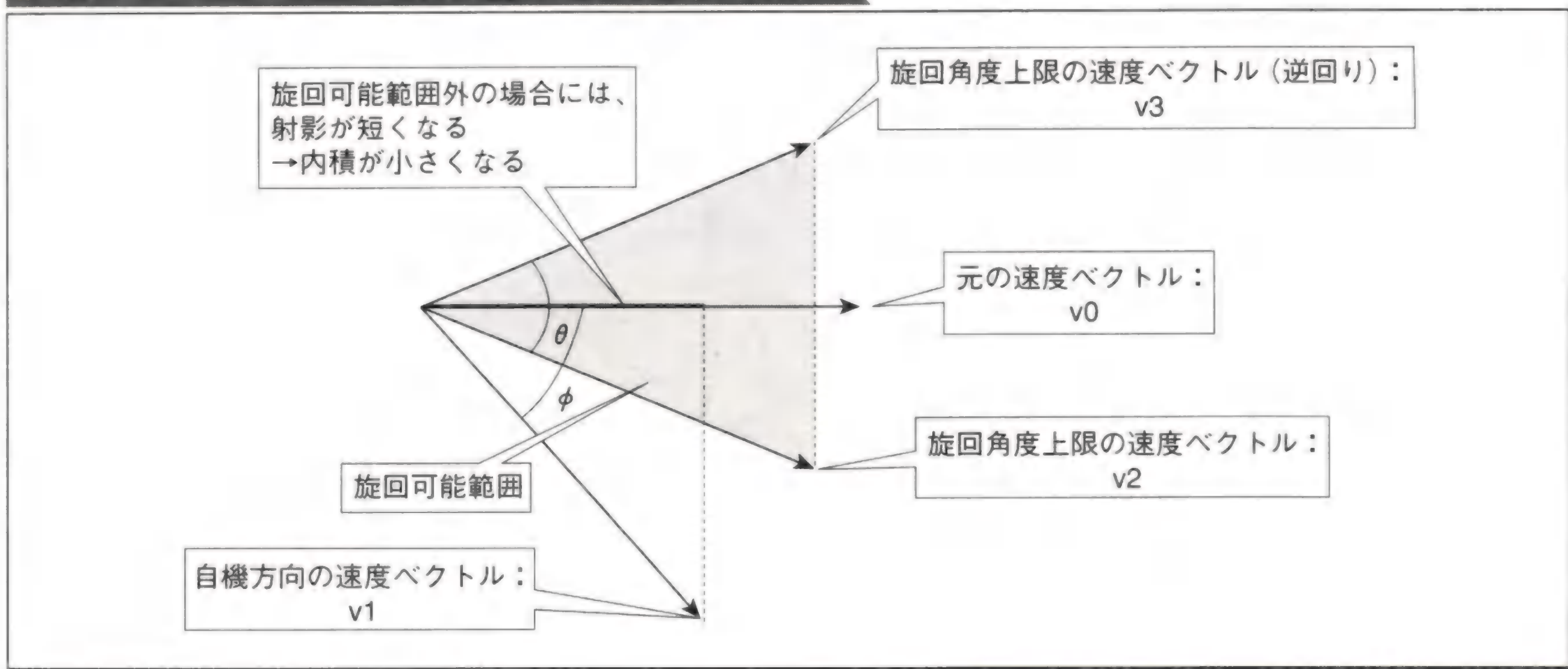
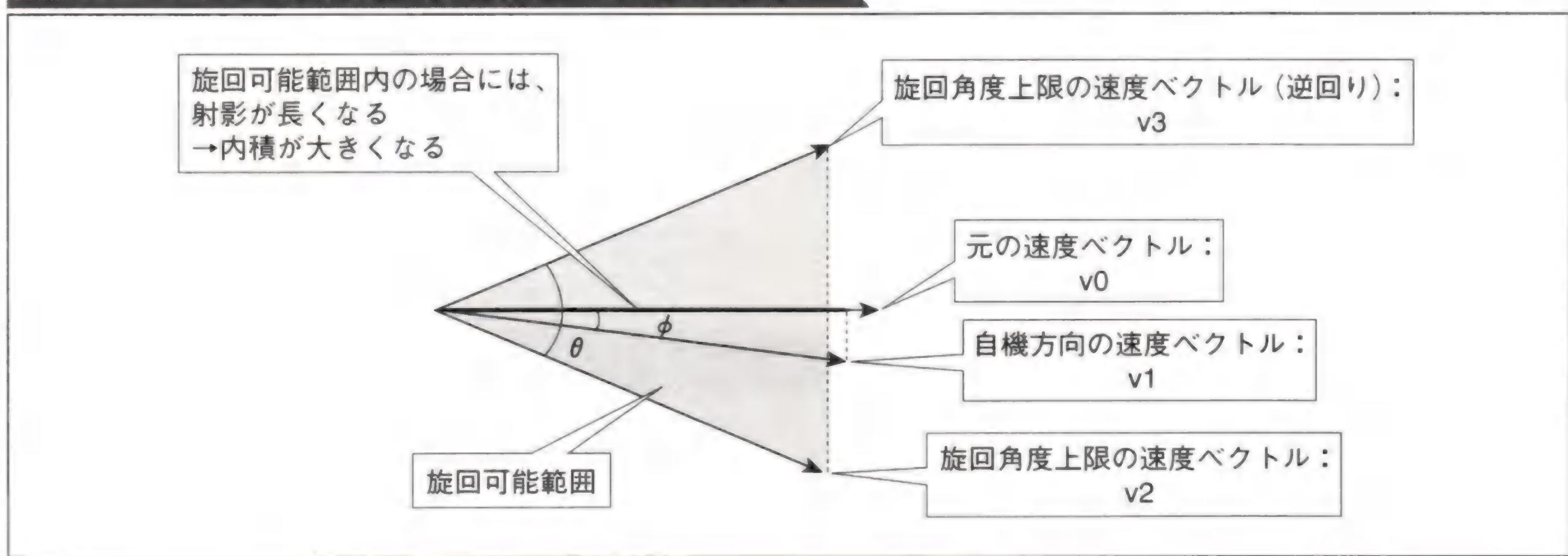



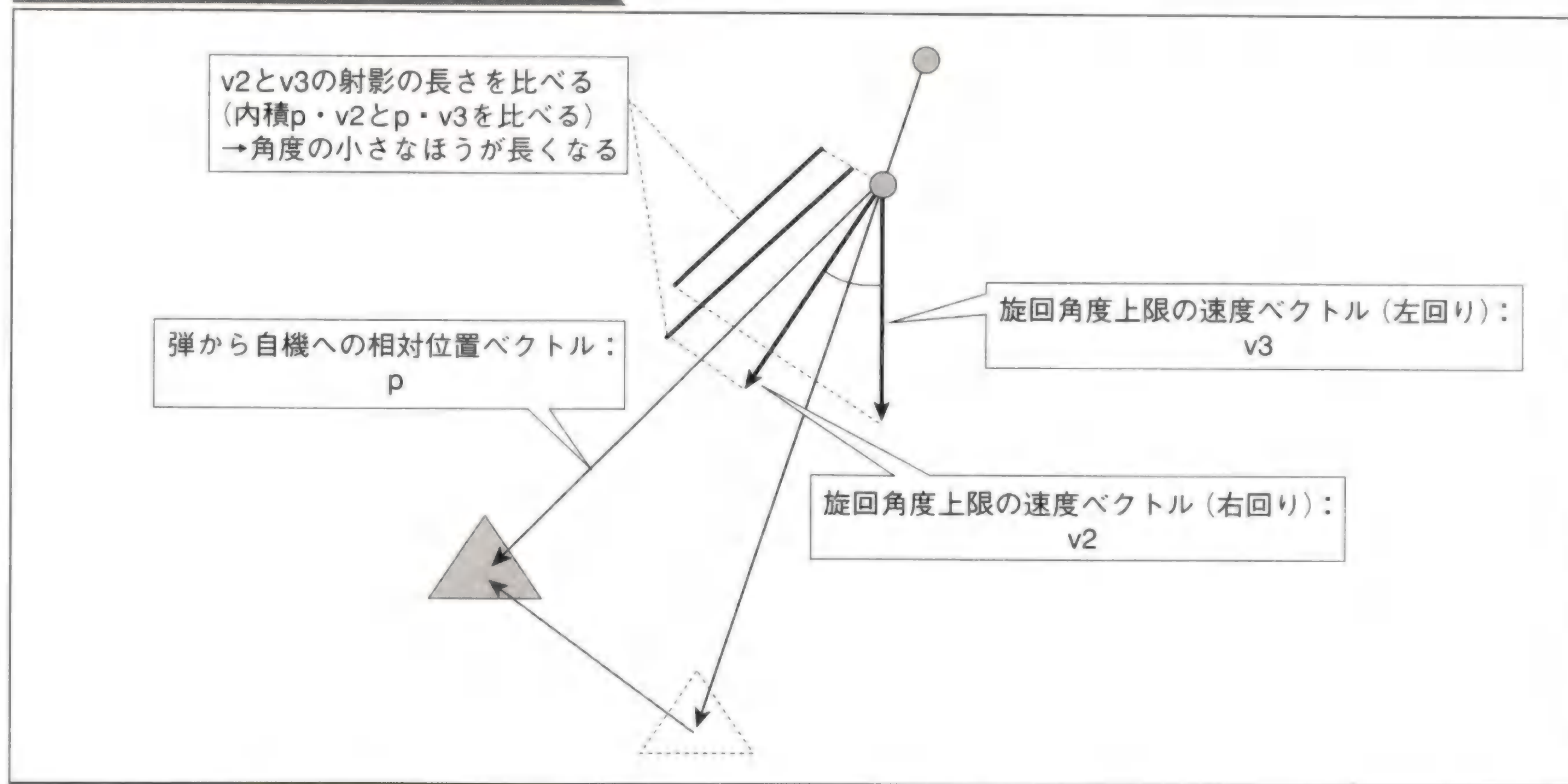
Fig. 2-43 旋回可能範囲内の場合には内積が大きくなる

これで自機の方向に曲がるか旋回角度の上限で曲がるかを決めることができますが、旋回角度の上限で曲がるときには、さらに右回りか左回りかを選ぶ必要があります。これに関してはFig. 2-44のように、やはり内積を使って決めることができます。

弾から自機への相対位置ベクトルをp、右回りの速度ベクトルをv2、左回りの速度ベクトルをv3とします。自機により近づく速度ベクトルというのは、v2とv3のうちpとなす角度が小さなベクトルです。これはpへの射影を比べればわかります。角度が小さなほうの射影が長くなるので、射影が長いほうの速度ベクトルを選べばよいのです。

Fig. 2-42や2-43と同様に、Fig. 2-44でも射影のかわりに内積を比べることができます。内積p・v2とp・v3を比べて、p・v2が大きければv2を選び、p・v3が大きければv3を選びます。

意外に手間がかかってしまいましたが、これで誘導弾の旋回角度を制限することができます。List 2-17は旋回角度を制限した誘導弾を動かすプログラムです。

旋回角度に上限がある誘導弾は、上手に回避すれば誘導弾を画面外に追い出すことができます。また、弾をうまく誘導すると自機の周りを回転させることも可能です。

Fig. 2-44 右回りか左回りかを選ぶ

サンプル

- 簡易誘導弾 → P. 313
- 誘導弾 → P. 313

List 2-17 旋回角度を制限した誘導弾の移動

```
#include <math.h>

void MoveHomingBullet(
    float& x, float& y,     // 弾の座標
    float& vx, float& vy,   // 弾の速度
    float mx, float my,     // 自機の座標
    float speed,            // 弾の速さ
    float theta             // 旋回角度の上限
) {
    // 弾の元の速度
    float vx0=vx, vy0=vy;

    // 自機方向の速度ベクトル(vx1, vy1)を求める
    float vx1, vy1;
    float d=sqrt((mx-x)*(mx-x)+(my-y)*(my-y));
    if (d) {
        vx1=(mx-x)/d*speed;
        vy1=(my-y)/d*speed;
    } else {
        vx1=0;
```

```
        vy1=speed;
    }

    // 右回り旋回角度上限の速度ベクトル(vx2, vy2)を求める：
    // M_PIは円周率。
    float rad=M_PI/180*theta;
    float vx2=cos(rad)*vx0-sin(rad)*vy0;
    float vy2=sin(rad)*vx0+cos(rad)*vy0;

     // 自機方向と旋回角度上限のどちらに曲がるかを決める
    if (vx0*vx1+vy0*vy1>=vx0*vx2+vy0*vy2) {

        // 自機方向が旋回可能範囲内の場合：
        // 自機方向に曲がる
        vx=vx1;
        vy=vy1;

    } else {

        // 自機方向が旋回可能範囲外の場合：
        // 左回り旋回角度上限の速度ベクトル(vx3, vy3)を求める。
        float vx3= cos(rad)*vx0+sin(rad)*vy0;
        float vy3=-sin(rad)*vx0+cos(rad)*vy0;

        // 弾から自機への相対位置ベクトル(px, py)を求める
        float px=mx-x, py=my-y;

        // 右回りか左回りかを決める
        if (px*vx2+py*vy2>=px*vx3+py*vy3) {

            // 右回りの場合
            vx=vx2;
            vy=vy2;

        } else {

            // 左回りの場合
            vx=vx3;
            vy=vy3;

        }
    }

    // 弾の座標(x, y)を更新して、弾を移動させる
    x+=vx;
    y+=vy;
}
```

誘導レーザー

自機を追尾するレーザーです（Fig. 2-45）。現実世界で「誘導レーザー」や「ホーミングレーザー」が物理的に実現可能かどうかはともかくとして、シューティングゲームのなかでは、こういった「曲がるレーザー」が広く使われています。見た目が派手で美しい弾です。

誘導レーザーは誘導弾（→ P. 39）の応用で作ることができます。実は、誘導レーザーの先端部分の動きは誘導弾の動きと同じです（Fig. 2-46）。つまり、先端部分に関しては誘導弾と同じアルゴリズムを適用できます。誘導弾と違うのは、誘導レーザーには尾があることです。

誘導レーザーの尾は先端部分が通った軌跡です。そこで、先端部分にもっとも近い尾の部分（尾1とします）は先端部分を追い、次に先端部分に近い尾の部分（尾2）は尾1を追い、さらに尾3は尾2を、尾4は尾3を……というように各部分が1つ前の部分を追いかけるようにすれば、尾を移動させることができます（Fig. 2-47）。

尾の各部分は自分の1つ前の部分を見て、1つ前の部分の古い座標へ移動するようにすればよ

Fig. 2-45 誘導レーザー

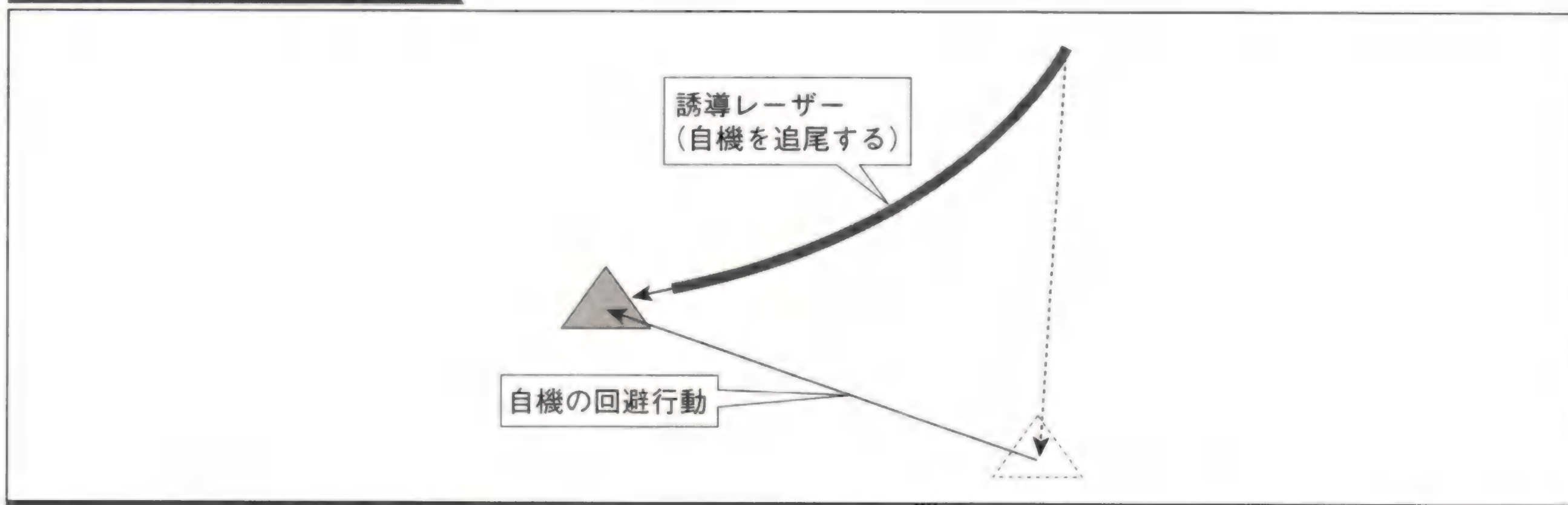



Fig. 2-46 誘導レーザーの仕組み

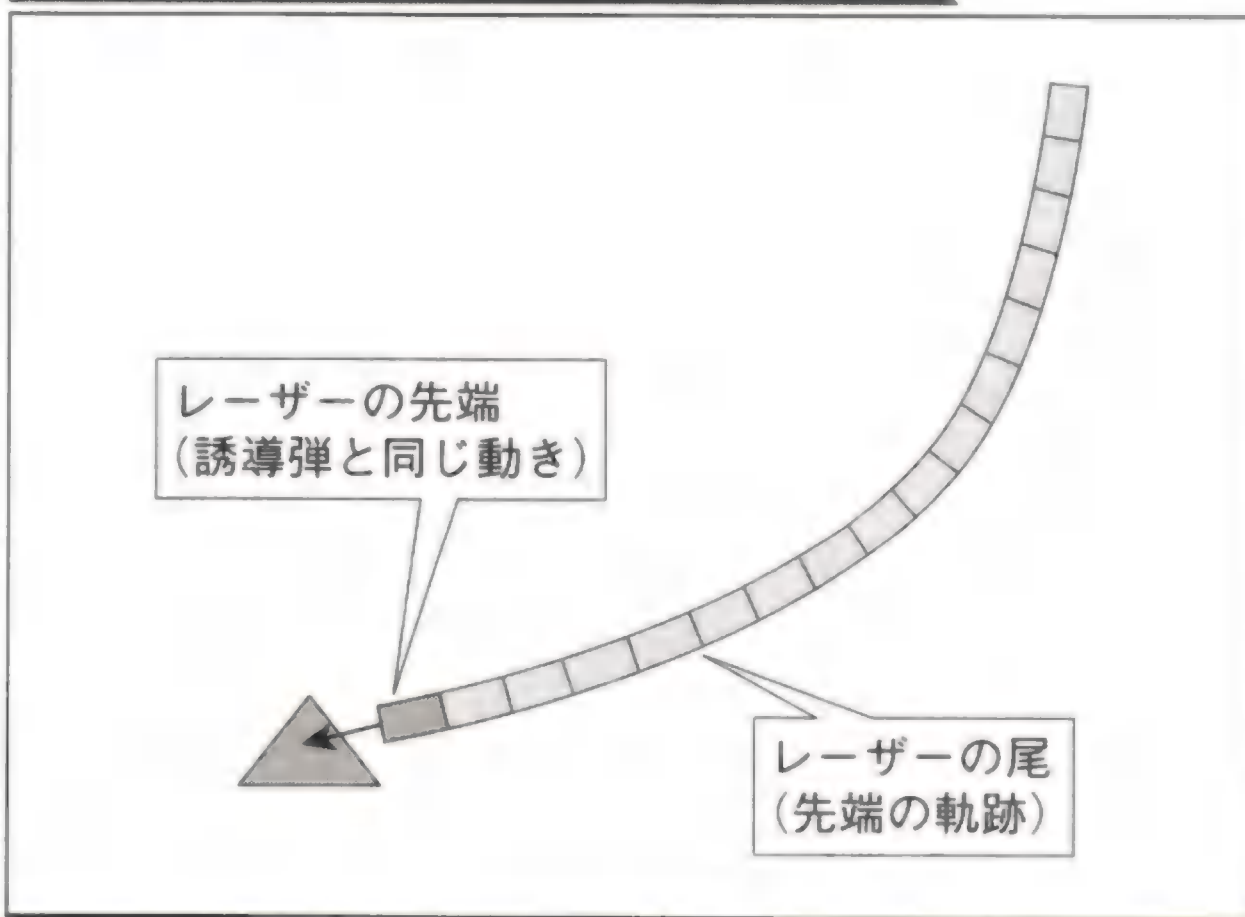
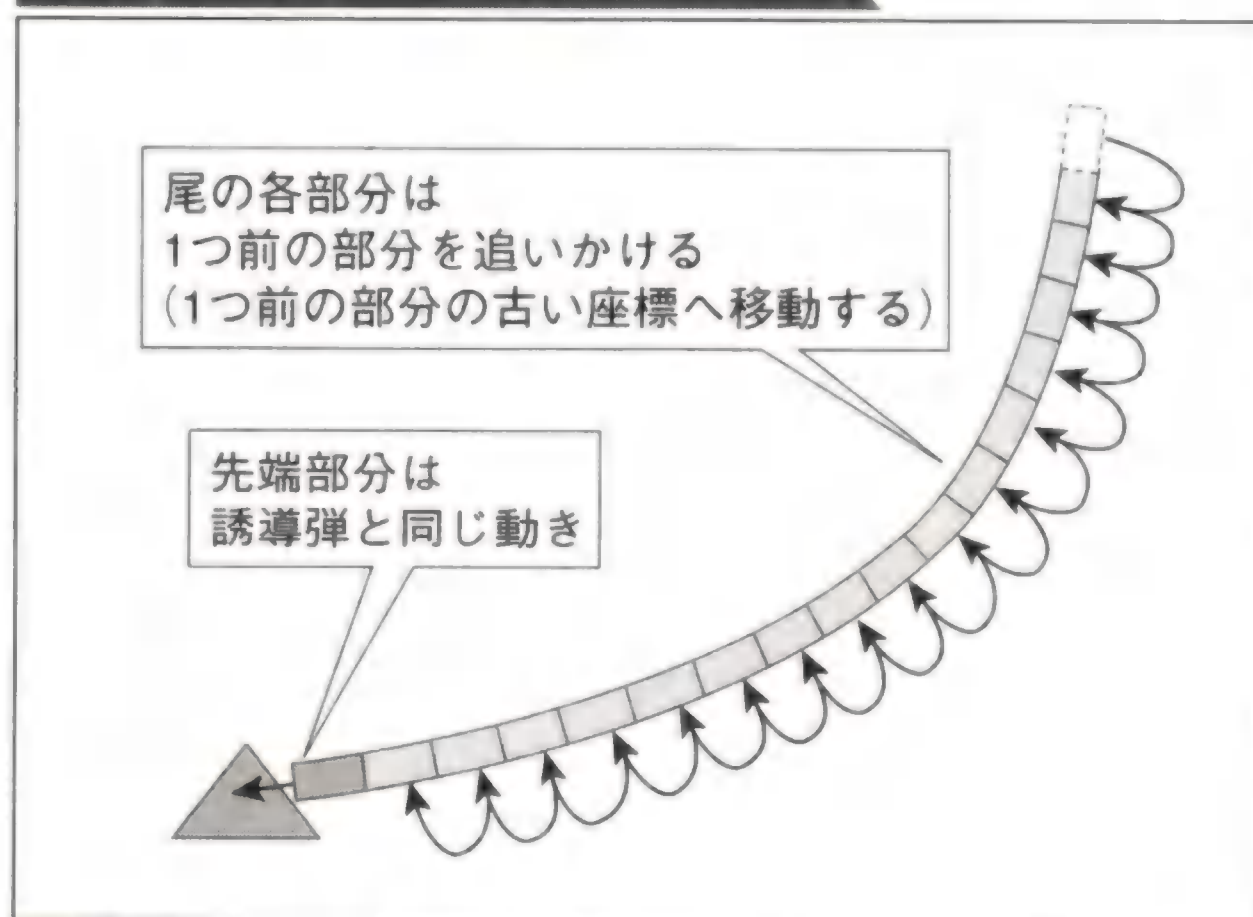
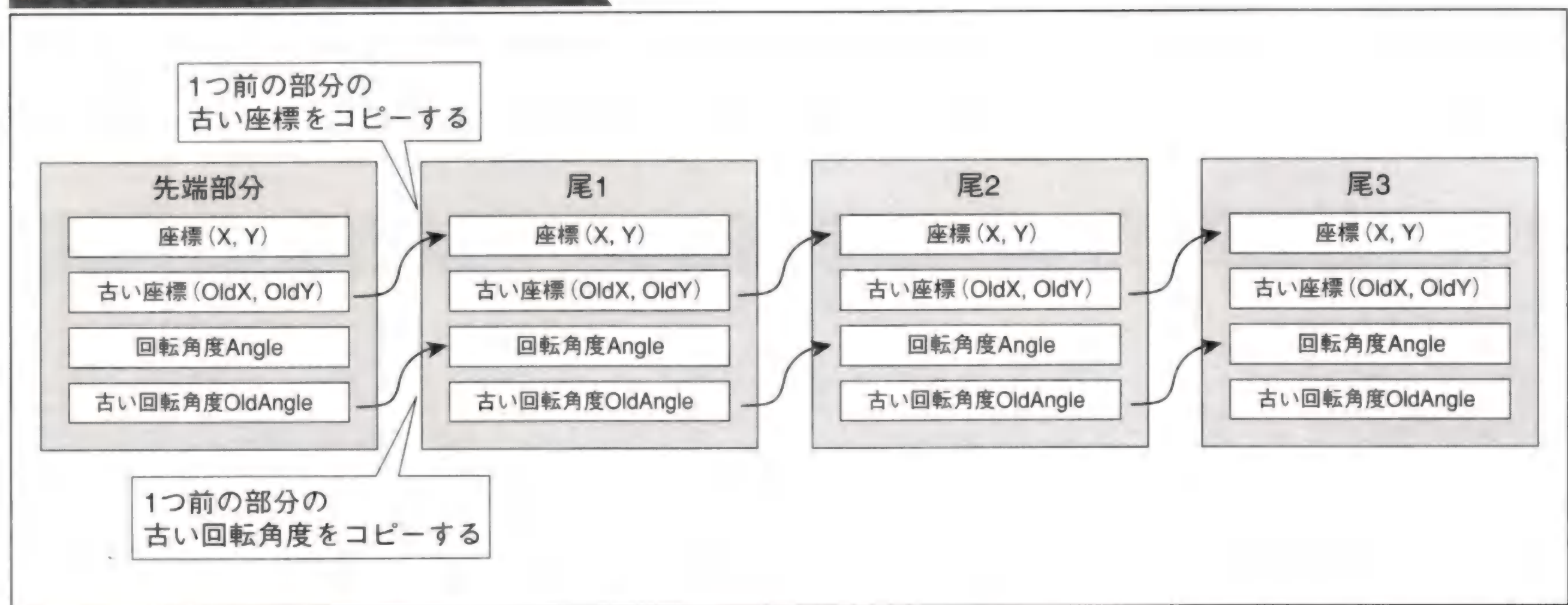



Fig. 2-47 誘導レーザーの尾

いのです。

具体的には、1つ前の部分の座標と回転角度をコピーして、次の部分の座標と回転座標にします（Fig. 2-48）。こうすれば、尾の各部分が1つ前の部分を追いかけるので、レーザー全体は先端部分が通った軌跡を描くようになります。

List 2-18は、ここで説明したような方法で誘導レーザーの移動を行うプログラムです。レーザーの先端部分は誘導弾と同じ方法で動かします。

List 2-19は誘導レーザーを発射するプログラムです。誘導レーザーの先端部分と尾の部分では、1つ前の部分があるかどうかが違います。先端部分の場合には1つ前をNULLとし、尾の場合には1つ前の部分を参照します。

Fig. 2-48 座標と回転角度のコピー

List 2-18 誘導レーザーの移動

```
#include <stdio.h>

// レーザーの各部分を表す構造体
typedef struct LASER_STRUCT {
    float X, Y;                  // 座標
    float VX, VY;                // 速度
    float OldX, OldY;            // 古い座標
    float Angle;                 // 回転角度
    float OldAngle;              // 古い回転角度
    struct LASER_STRUCT* Prec;   // 1つ前の部分へのポインタ
                                 // （先頭部分の場合にはNULL）
} LASER_TYPE;

// レーザーを動かす関数
void MoveHomingLaser(
    LASER_TYPE* laser  // レーザーの先頭部分または尾の一部分
) {
```

```
    // 先端部分の場合：
    // 誘導弾の動きと同じ。
    // 誘導の具体的な処理はMoveHoming関数で行うとする。
    if (laser->Prec==NULL) {
        MoveHoming(laser);
    }

    // 先端部分以外の場合：
    // 1つ前の部分を追いかける。
    // 古い座標と古い回転角度をコピーする。
    else {
        laser->X=laser->Prec->OldX;
        laser->Y=laser->Prec->OldY;
        laser->Angle=laser->Prec->OldAngle;
    }
}
```

List 2-19 誘導レーザーの発射

```
#include <stdio.h>

// レーザーの各部分を表す構造体
typedef struct LASER_STRUCT {
    float X, Y;                      // 座標
    float VX, VY;                    // 速度
    float OldX, OldY;                // 古い座標
    float Angle;                     // 回転角度
    float OldAngle;                  // 古い回転角度
    struct LASER_STRUCT* Prec;       // 1つ前の部分へのポインタ
                                     // (先頭部分の場合にはNULL)
} LASER_TYPE;

// レーザーの発射
void ShootHomingLaser(
    float x, float y,    // 発射地点の座標
    int length           // レーザーの長さ
) {
    LASER_TYPE* laser;        // レーザーを表す構造体へのポインタ
    LASER_TYPE* prec=NULL;    // 1つ前の部分を指すポインタ

    // レーザーの各部分を作る：
    // レーザーの構造体を確保し、座標を初期化する。
    // 構造体確保の具体的な処理はNewLaserType関数で行うとする。
    for (int i=0; i<length; i++, prec=laser) {
        laser=NewLaserType();
        laser->X=laser->OldX=x;
```

```
        laser->Y=laser->OldY=y;
        laser->Angle=laser->OldAngle=0;

        // 尾の部分は1つ前の部分を参照するようにし、
        // 先端部分にはNULLを参照させる
        laser->Prec=prec;
    }
}
```

誘導レーザーの描画

レーザーの先端部分と尾の各部分を描くには、Fig. 2-49-①のような四角形を使うことも、Fig. 2-49-②のように角が丸い図形を使うこともできます。レーザーの先端を丸くしたいときには、Fig. 2-49-②のように角を丸くしておきます。

各部分を描くために使う四角形の太さは、レーザーの太さに応じて決めます。四角形を太くすればレーザーも太くなり、四角形を細くすればレーザーも細くなります。一方、四角形の長さについては、レーザーの速さに応じて決める必要があります(Fig. 2-50)。

Fig. 2-49 レーザーの描画方法

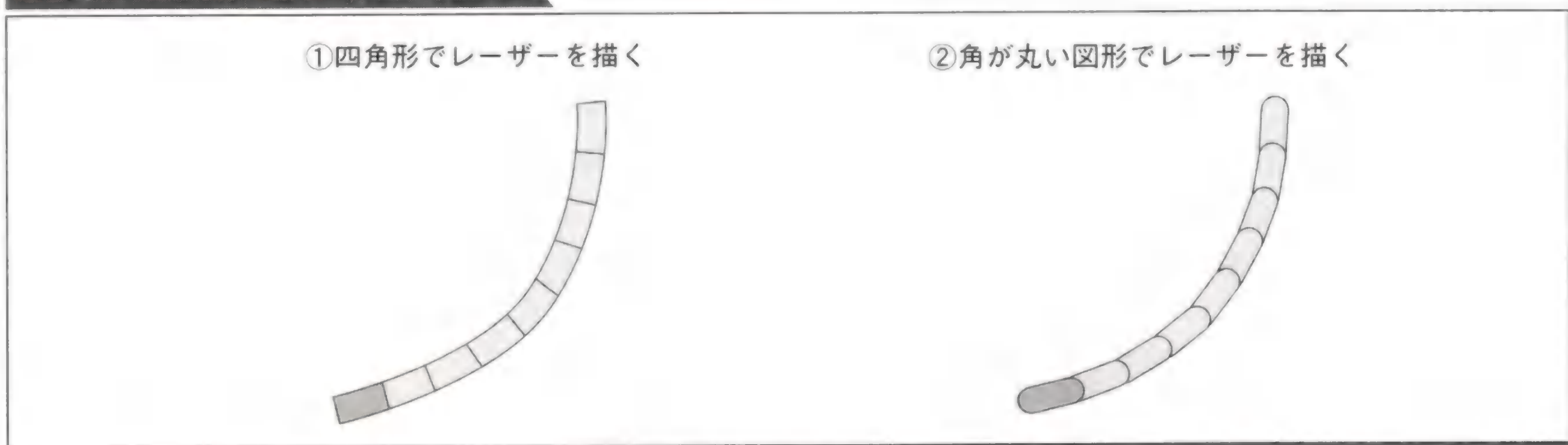



Fig. 2-50 四角形の長さとレーザーの速さ

レーザーが速いのに四角形が短いと、Fig. 2-50-②のようにレーザーがとぎれてしまいます。逆にレーザーが遅いのに四角形が長いと、Fig. 2-50-③のように四角形の角が見えてしまいます。

なめらかなレーザーを描くには、Fig. 2-50-①のようにレーザーの速さと四角形の長さとのバランスをとらなければなりません。多少バランスが崩れたくらいならば、レーザーの太さが微妙に変わるだけなので大丈夫ですが、あまり極端に崩れると問題になります。実際にレーザーを動かしながら、レーザーを構成する四角形の長さを調整するとよいでしょう。

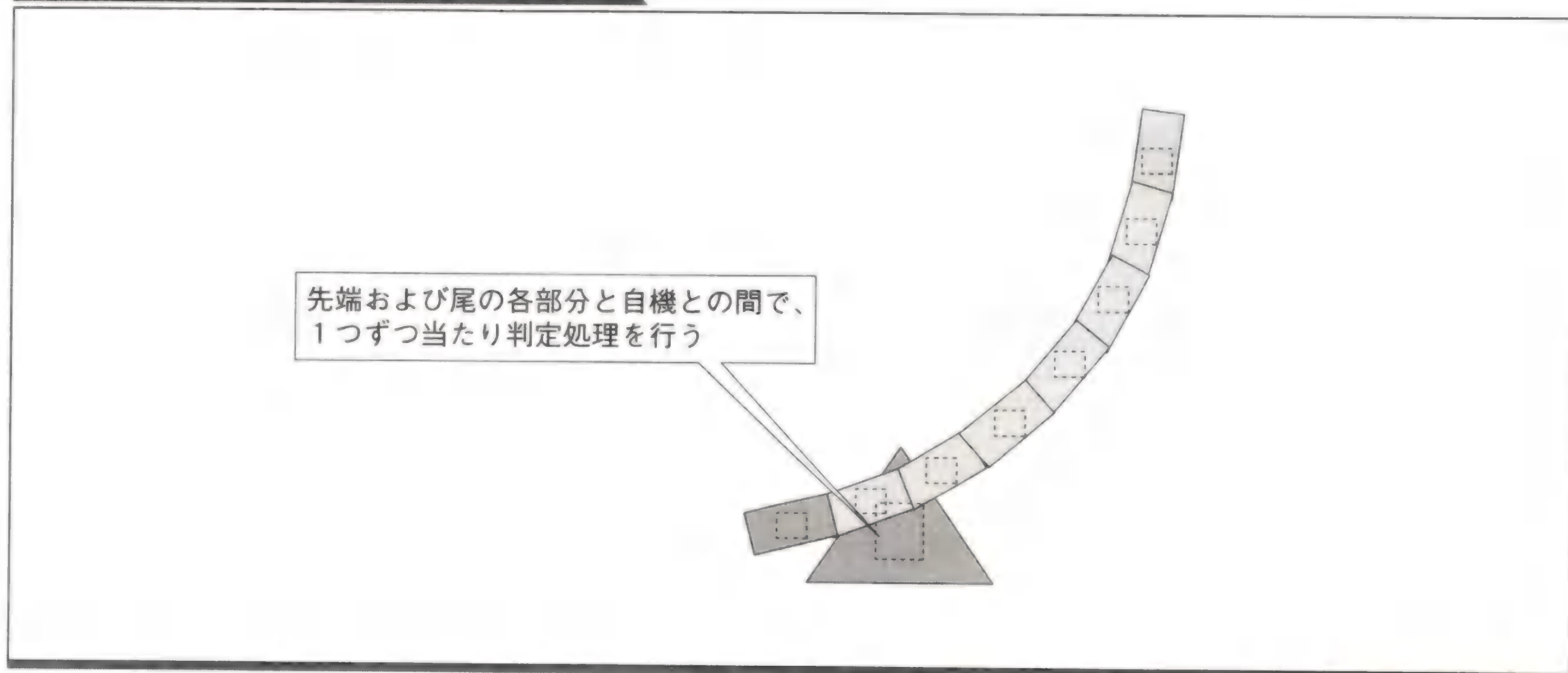
※

レーザーを描く際には、アルファ合成（アルファブレンディング）や加算合成を使うのがお勧めです。特に加算合成を使うと、光源っぽい質感でレーザーを描くことができます。レーザーが重なった部分が明るく輝くので非常にきれいです。アルファ合成や加算合成は利用するグラフィックライブラリの機能によって使い方が違うので、詳しくはお使いのライブラリのマニュアルを調べてみてください。なお、DirectXはアルファ合成も加算合成もサポートしています。

■ 誘導レーザーの当たり判定

レーザーは普通の弾とは違って、尾にも当たり判定があります。そのため、レーザーの当たり判定処理を行うには、先端だけでなく尾と自機との間でも判定をしなければなりません（Fig. 2-51）。

レーザーの各部分は回転（角度を変える）するので、厳密には当たり判定もFig. 2-52のように回転させる必要があります。しかし、実際にはFig. 2-51のように当たり判定を回転させないままでも、それほど問題にはなりません。回転させないほうが処理はずっと簡単なので、問題がなければ回転を省略してもかまわないでしょう。

Fig. 2-51 誘導レーザーの当たり判定

回転を省略する場合には、当たり判定を正方形にしておく必要があります。さもないとFig. 2-53のように、表示が回転したときに当たり判定がレーザーの軌跡からはみ出してしまい、不自然な当たり判定になってしまう恐れがあります。

※

誘導レーザーの先端部分の動きは誘導弾と同じですが、レーザーにするとずいぶんと見た目が派手になります。レーザーは尾の部分にも当たり判定があるので、ゲーム性の点でも誘導弾とはだいぶ違います。

誘導レーザーの動きに変化をつけると、さらに面白くなります。たとえば、発射された直後には誘導を行わずに、一定時間が経過したあとに誘導を始めると、「直進したあとに曲がるレーザー」を作ることができます。

Fig. 2-52 当たり判定を回転させる

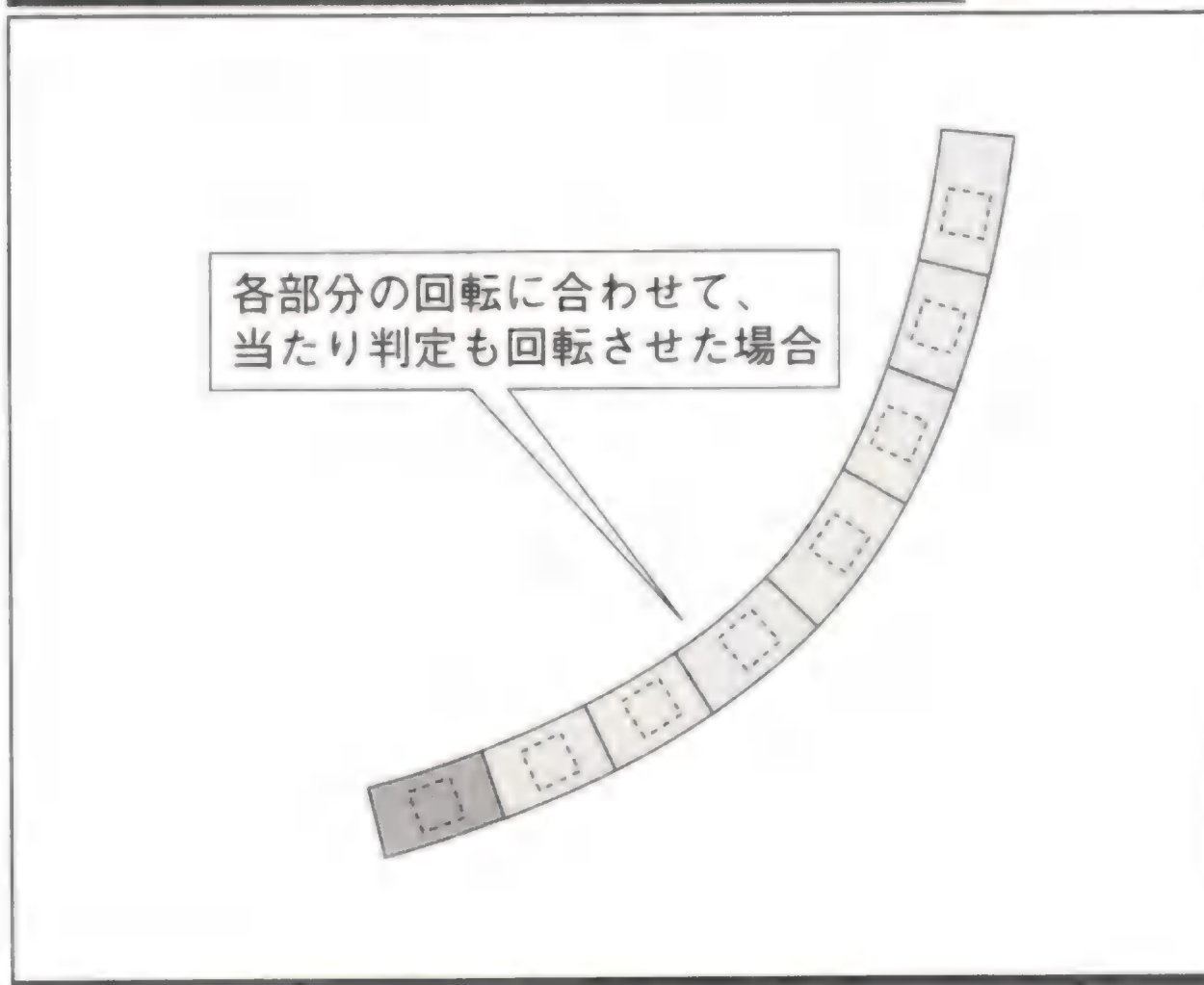
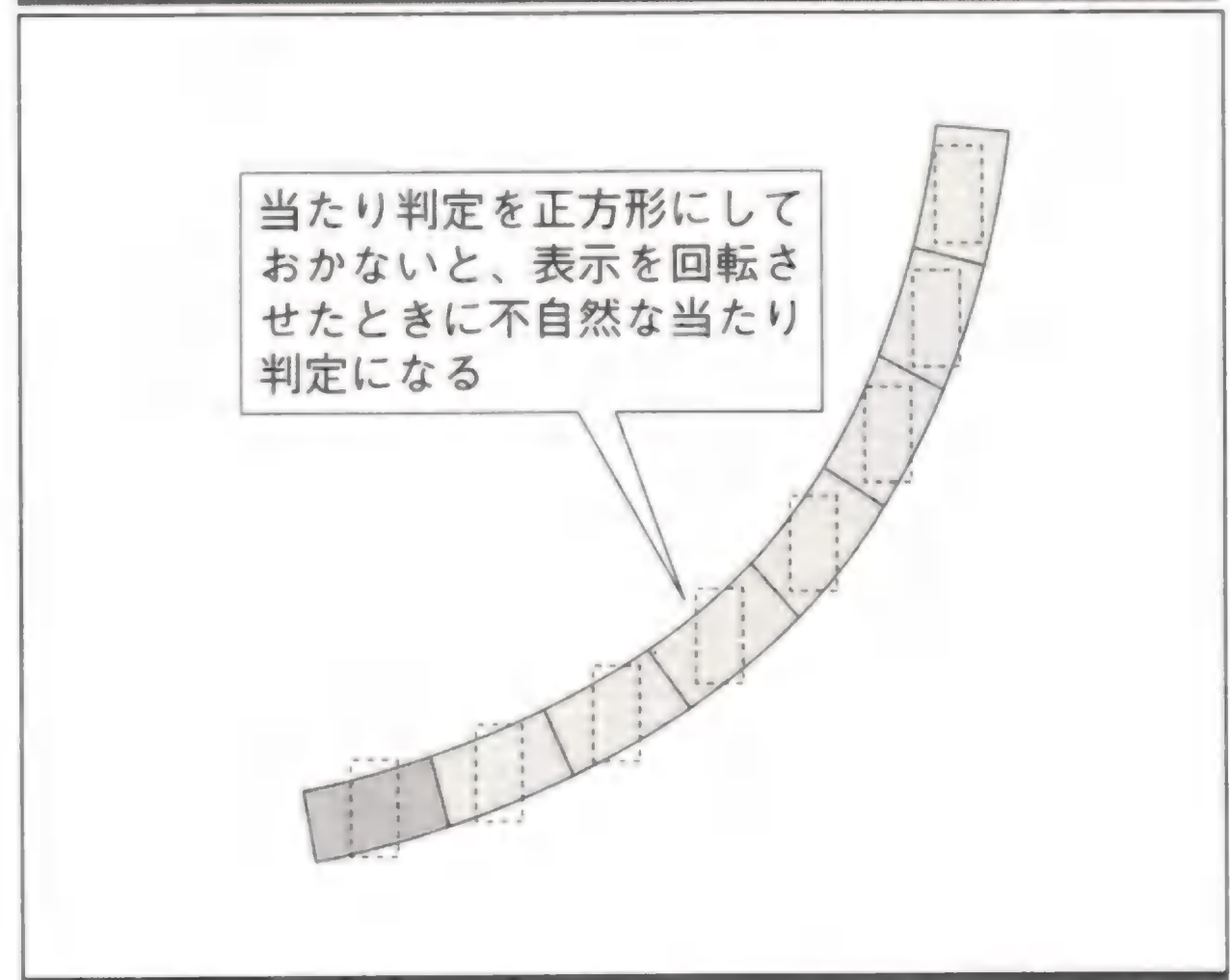
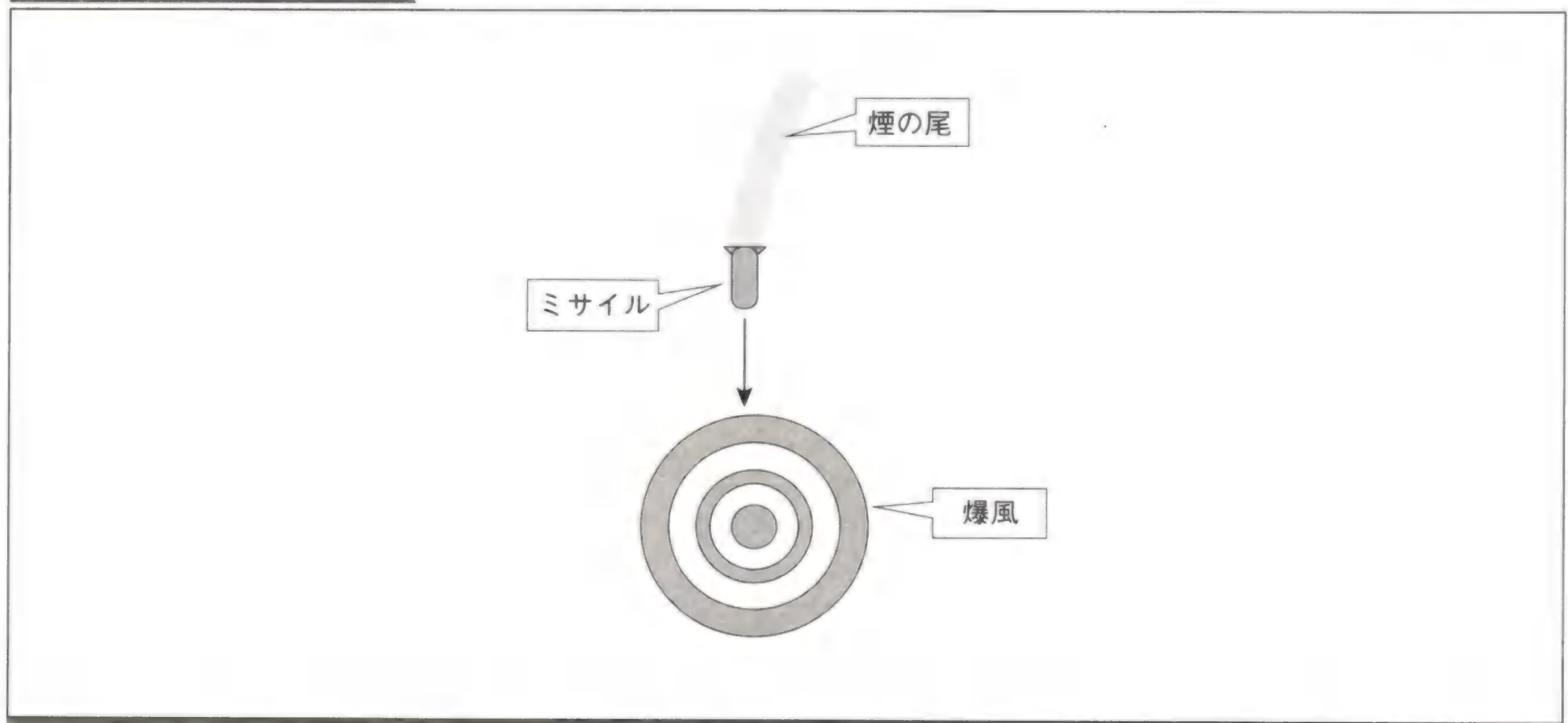



Fig. 2-53 正方形ではない当たり判定を使ったときの問題

サンプル

- 誘導レーザー　→ P. 313
- 誘導レーザー2　→ P. 313

ミサイル

ミサイルといってもさまざまなものが考えられますが、ここでは「煙の尾を引きながら飛行して、爆発するミサイル」を考えます（Fig. 2-54）。当たり判定はミサイル本体と爆風にあることにします。つまり、プレイヤーはミサイル本体と爆風を回避しなければならないわけです。煙の尾には当たり判定がないので、よける必要はありません。

Fig. 2-54 ミサイル

煙の処理はめんどうそうですが、実は誘導レーザー（→P.46）の応用で作ることができます。レーザーでは先端部分と同じパターンを使って尾を描きますが、ミサイルでは先端部分にミサイルを描き、尾の部分には煙を描けばよいのです（Fig. 2-55）。煙の表現にアルファ合成を使ってみても面白いでしょう。

ミサイルを爆発させるタイミングはいろいろと考えられます。たとえば、発射されてから一定時間が経過したら爆発するとか、自機までの距離が一定値以下になったら爆発するなどといった方法があります。ミサイルを爆発させるときには、ミサイルを消して、かわりに爆風を出現させます（Fig. 2-56）。爆風は時間とともに大きくして、一定時間を超えたら消すようにします。爆風のかわりに弾をばらまくミサイルなども、同じ要領で作ることができます。

List 2-20はミサイル発射のプログラムです。

サンプル

誘導ミサイル　→ P. 313

Fig. 2-55 ミサイルと煙の描画方法

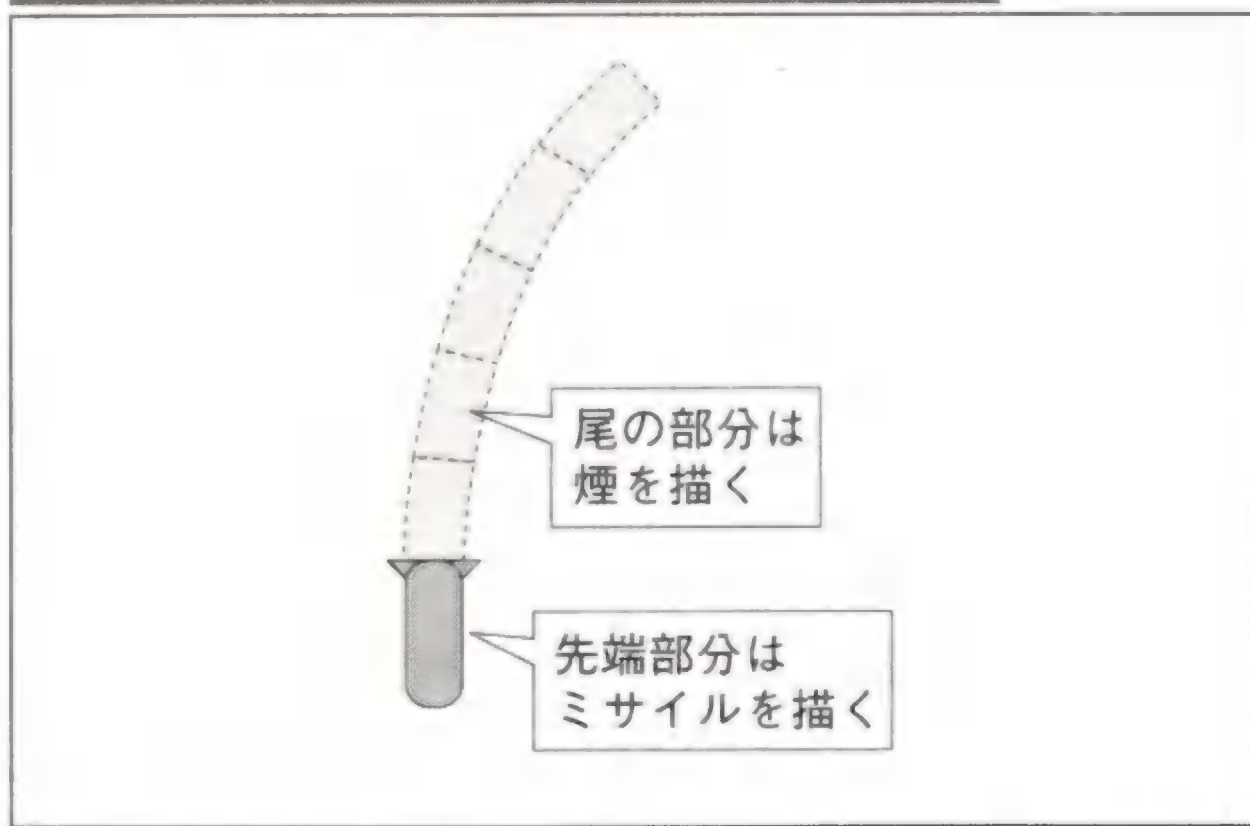
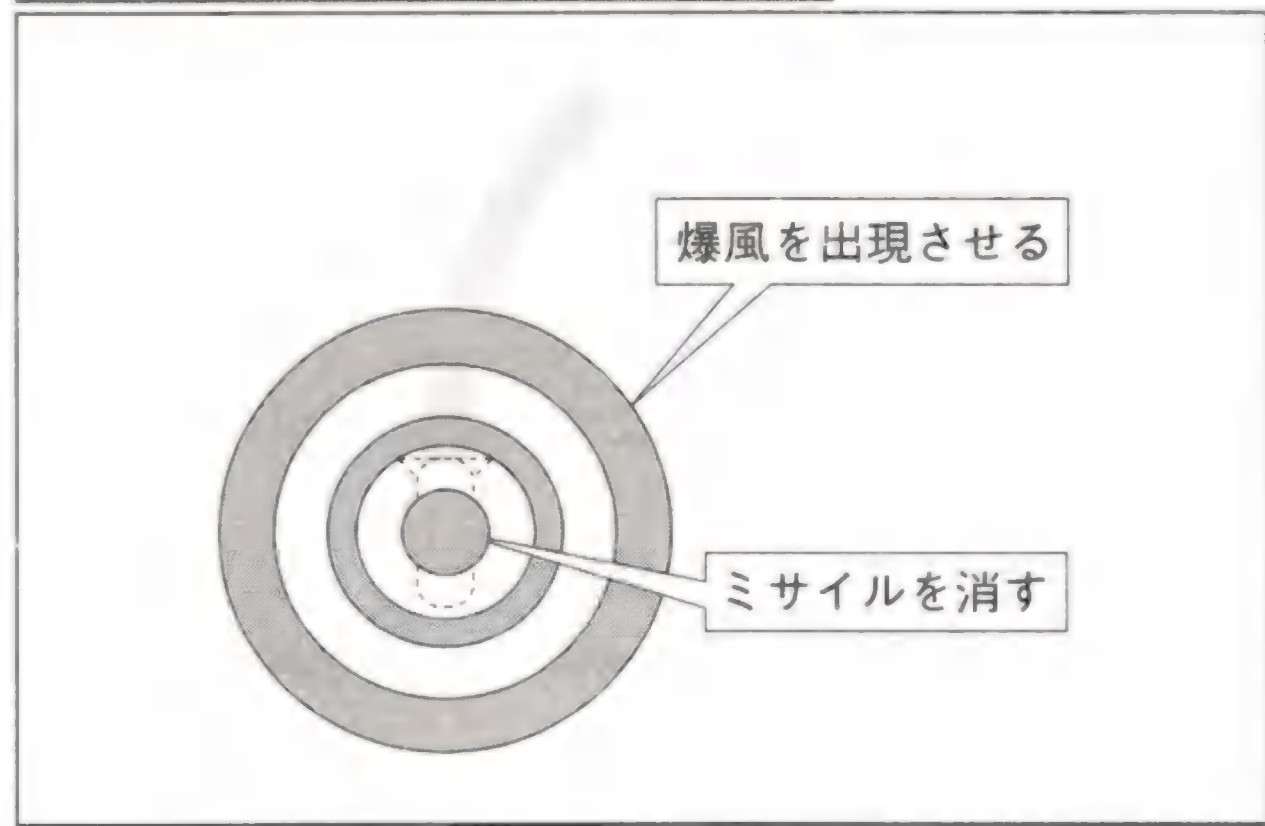



Fig. 2-56 ミサイルの爆発

List 2-20 ミサイルの発射

```
#include <stdio.h>

// ミサイルの各部分を表す構造体
typedef struct MISSILE_STRUCT {
    float X, Y;                     // 座標
    float VX, VY;                   // 速度
    float OldX, OldY;               // 古い座標
    float Angle;                    // 回転角度
    float OldAngle;                 // 古い回転角度
    struct MISSILE_STRUCT* Prec;    // 1つ前の部分へのポインタ
                                    // (先頭部分の場合にはNULL)
    bool IsMissile;                 // ミサイルかどうか
                                    // (trueならミサイル、
                                    //  falseなら煙)
} MISSILE_TYPE;

// ミサイルの発射
void ShootMissile(
    float x, float y,   // 発射地点の座標
    int length          // レーザーの長さ
) {
    MISSILE_TYPE* missile;      // ミサイルの構造体へのポインタ
    MISSILE_TYPE* prec=NULL;    // 1つ前の部分を指すポインタ

    // ミサイルの各部分を作る:
    // ミサイルの構造体を確保し、座標を初期化する。
    // 構造体確保の具体的な処理はNewMissileType関数で行うとする。
    for (int i=0; i<length; i++, prec=missile) {
        missile=NewMissileType();
        missile->X=missile->OldX=x;
        missile->Y=missile->OldY=y;
        missile->Angle=missile->OldAngle=0;
```

```
        // 尾の部分は1つ前の部分を参照するようにし、
        // 先端部分にはNULLを参照させる
        missile->Prec=prec;

        // 先端部分はミサイルに、ほかの部分は煙にする
        missile->IsMissile=(i==0);
    }
}
```

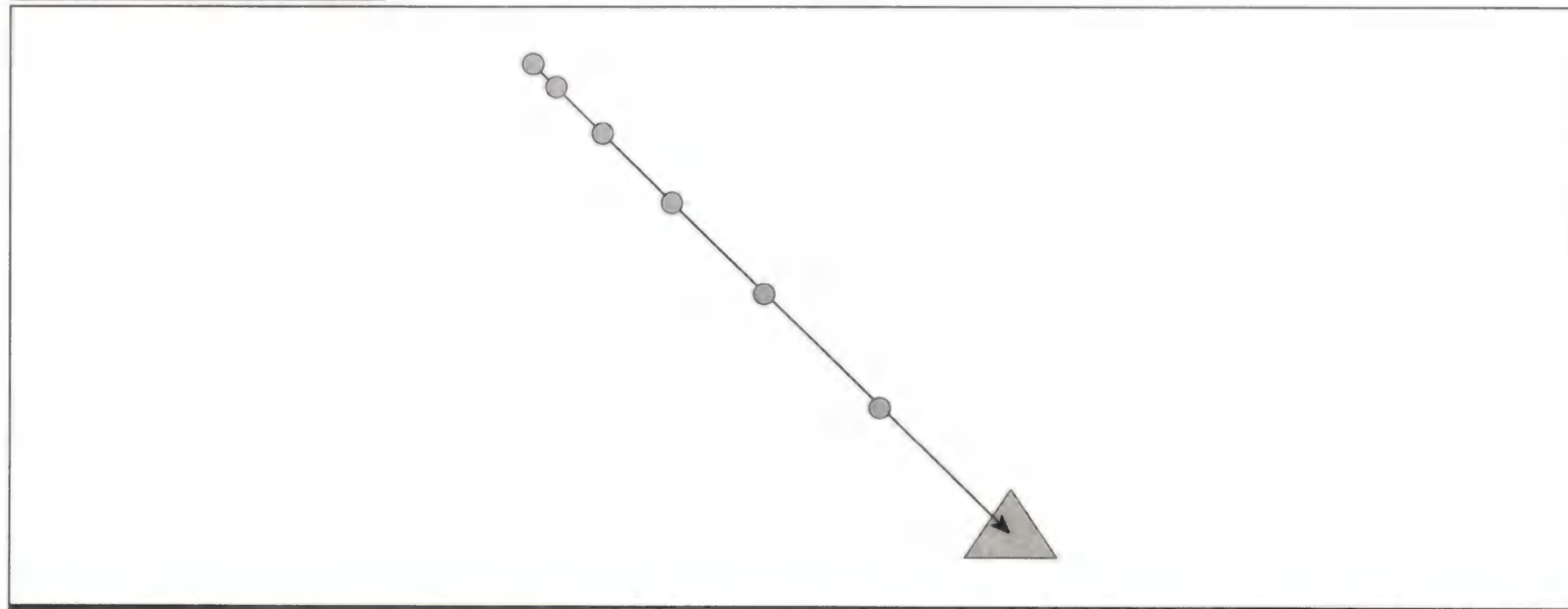
加速弾

発射されたあとに速度が変わる弾です（Fig. 2-57）。だんだん速くなる弾もあれば、逆にだんだん遅くなる弾もあります。速さが0になったり逆方向に進んだりする弾はあまり見かけませんが、それらも加速弾の一種です。

弾を加速させるには、弾を動かすたびに弾の速度に対して一定の値（加速度）を加えます。これは物理学でいう「等加速度運動」です。たとえば最初の速度（初速度）をspeed、加速度をaccelとすると、弾の速度はFig. 2-58のように変化します。

このように加速弾では、弾を動かすたびに速度に対して加速度を加えます。一般に、速度speedの弾を加速度accelでn回動かしたあとの速度speed_newは次の式で求められます。

speed_new=speed+accel*n

Fig. 2-57 加速弾

速度（vx, vy）、速さspeedで直進する弾の場合、速さspeedが変わったときには速度（vx, vy）を次のように更新します。速度の各成分を古い速さで割り、新しい速さを掛けます。

vx=vx/speed*speed_new
vy=vy/speed*speed_new

List 2-21は速さspeedと速度（vx, vy）を更新するプログラムです。速度の各成分を速さで割り、速さに加速度を加えたあとに速度に新しい速さを掛けます。

Fig. 2-58 加速度による速度の変化

速度speed
速度speed+accel
速度speed+accel*2
速度speed+accel*3
速度speed+accel*4
速度speed+accel*5

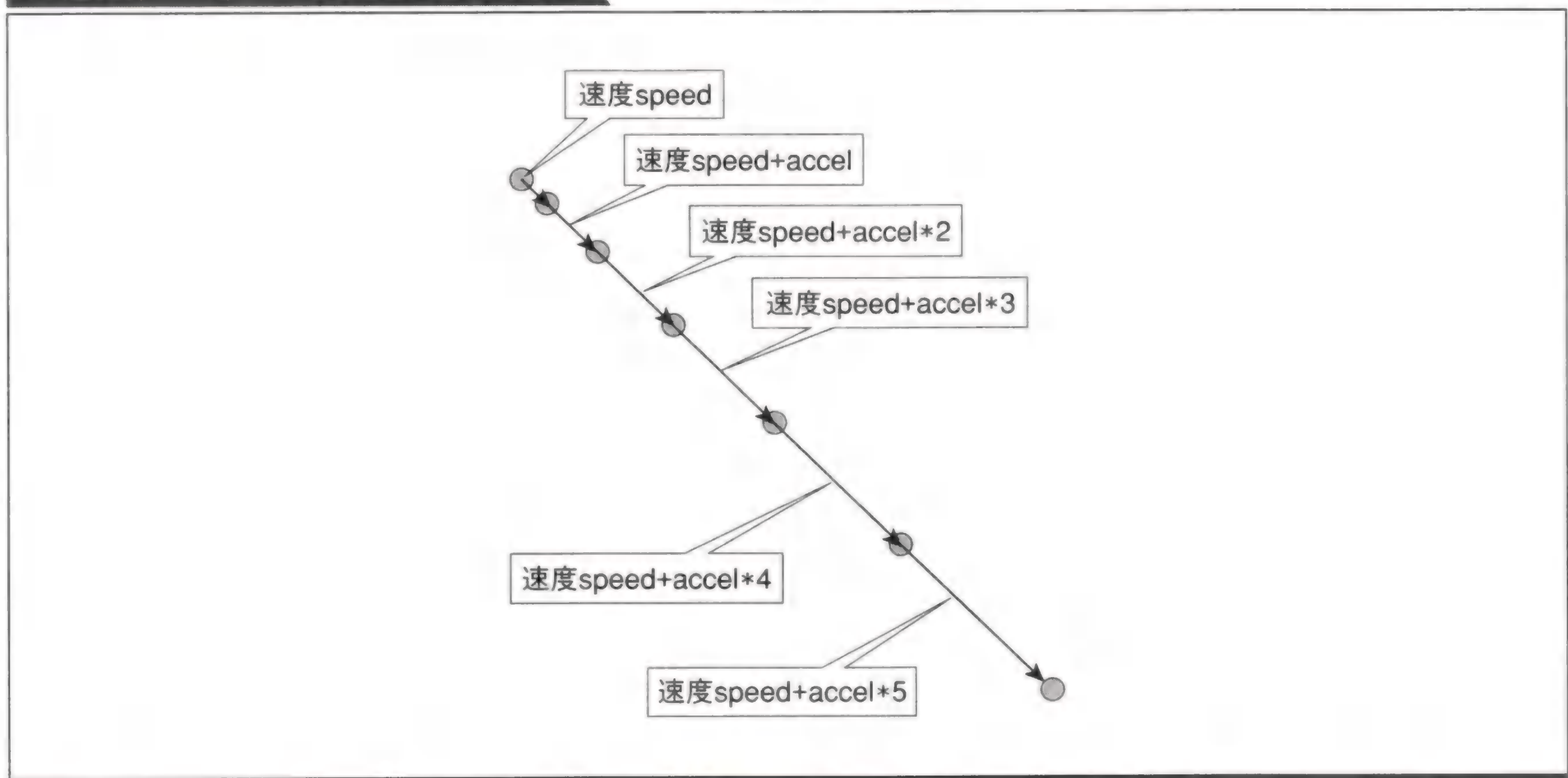



Fig. 2-58のように直進する弾だけではなく、誘導弾のように曲がる弾を加速させることもできます。誘導弾が加速したり減速したりすると、ゲームとしても面白くなります。たとえば、一定速度では誘導がきつすぎる（よけられない）誘導弾でも、加速したり減速したりすればよけられるようになります。誘導レーザーと加速を組み合わせるのもよい方法です。速くなったり遅くなったりしながら進むレーザーの動きは独特で、見た目にも非常に楽しくなります。

加速するレーザーのバリエーションでは「レイフォース（→ P. 335）」「レイストーム（→ P. 335）」が大いに参考になります。特に「レイフォース」では、古典的な2Dグラフィックにもかかわらずバリエーション豊かなレーザーを実現していることが驚きです。

サンプル

● 加速n-way弾 → P. 313
● 加速誘導レーザー → P. 313

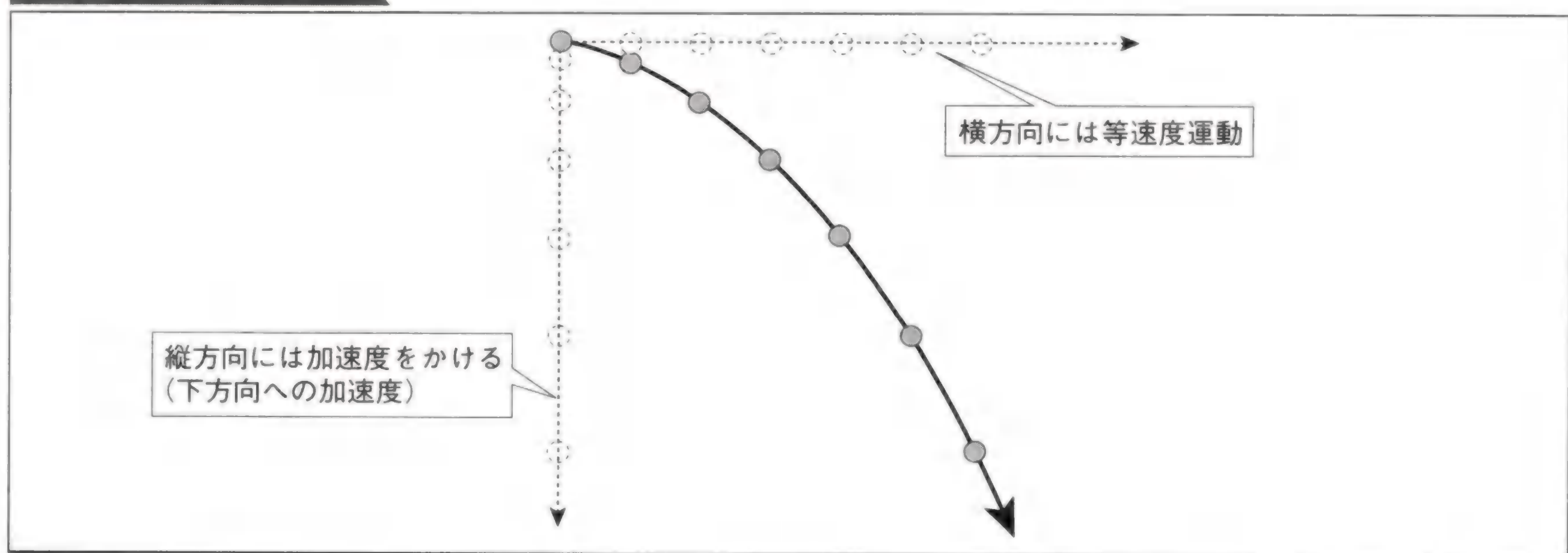
List 2-21 速さと速度の更新

```
void MoveAcceleratedBullet(
    float speed,          // 弾の速さ
    float accel,          // 加速度
    float& vx, float& vy  // 速度のX成分とY成分
) {
    // 速度を古い速さで割る
    if (speed!=0) {
        vx/=speed; vy/=speed;
    }

    // 速さを更新する
    speed+=accel;

    // 速度に新しい速さを掛ける
    vx*=speed; vy*=speed;
}
```

落下弾

加速弾 (→ P. 54) を応用すると、放物線を描いて落下する弾を作ることができます。このような弾は縦スクロールゲームではあまり見かけませんが、横スクロールゲームではときどき見かけます。爆弾や手榴弾などを投げるゲームでは、投げた物体が放物線を描きながら落下します。

放物線を描く弾のアルゴリズムは簡単です。縦方向に加速度を掛けて、横方向は等速度で移動させれば、弾の軌跡は放物線になります (Fig. 2-59)。

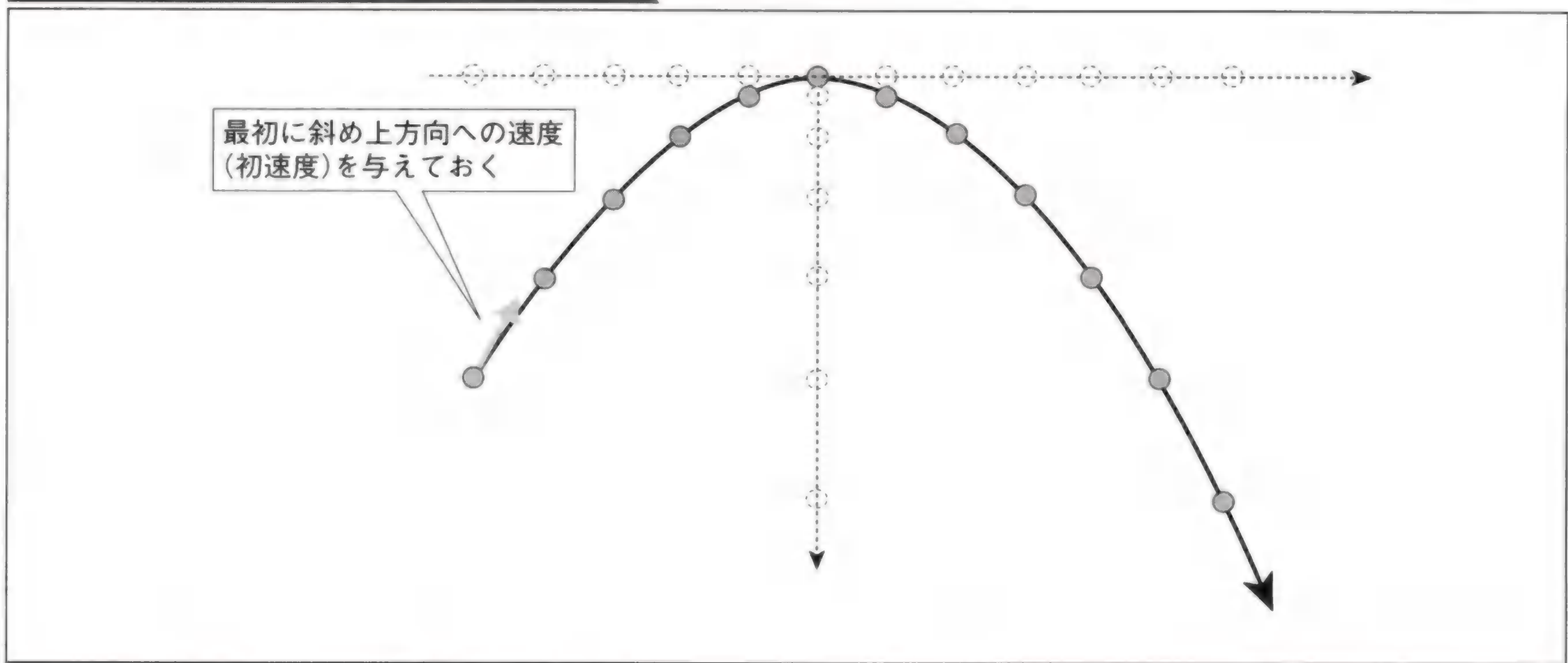
Fig. 2-59 落下弾

最初に斜め上方向への速度を与えておけば、弓なりに上昇してから落下する弾を作ることもできます (Fig. 2-60)。

List 2-22は落下弾を移動させるプログラムです。

なお、ミサイルに落下を組み合わせるのはお勧めです。ミサイルの尾が放物線を描くので、面白い効果が得られます。

サンプル

落下弾 → P. 313
落下ミサイル → P. 314

Fig. 2-60 上昇してから落下する落下弾

List 2-22 落下弾の移動

```
void MoveDroppingBullet(
    float& x, float& y,    // 弾の座標
    float accel,           // 加速度

    float& vx, float& vy   // 速度のX成分とY成分
) {
    // 速度を更新する：
    // X成分は変化させず、Y成分だけを変化させる
    vy+=accel;

    // 座標を更新する
    x+=vx; y+=vy;
}
```

回転弾

中心点の周囲を回転する弾です(Fig. 2-61)。この類の弾はボスキャラなどでよく見かけます。回転半径や回転速度が異なる数種類の弾を組み合わせることもよくあります(Fig. 2-62)。

回転弾の座標はFig. 2-63のように計算します。弾を移動させるには、移動のたびに角度θ(単位はラジアン)を変化させます。1回の移動で変化する角度をω(ラジアン)とすると、新しい角度は$\theta+\omega$です。この方法で弾を移動させるプログラムはList 2-23のようになります。

List 2-23では弾の速度(vx, vy)も求めています。速度を求めるには、Fig. 2-64のように弾の角速度$r*\omega$を求め、これに$-\sin\theta$と$\cos\theta$を掛けてX方向とY方向の成分に分解します。弾の速度は必要なときだけ求めればよいでしょう。たとえば針状の弾やレーザーなどでは、弾が進む向きによってグラフィックを変えたいので、速度を計算する必要があります。

Fig. 2-61 回転弾

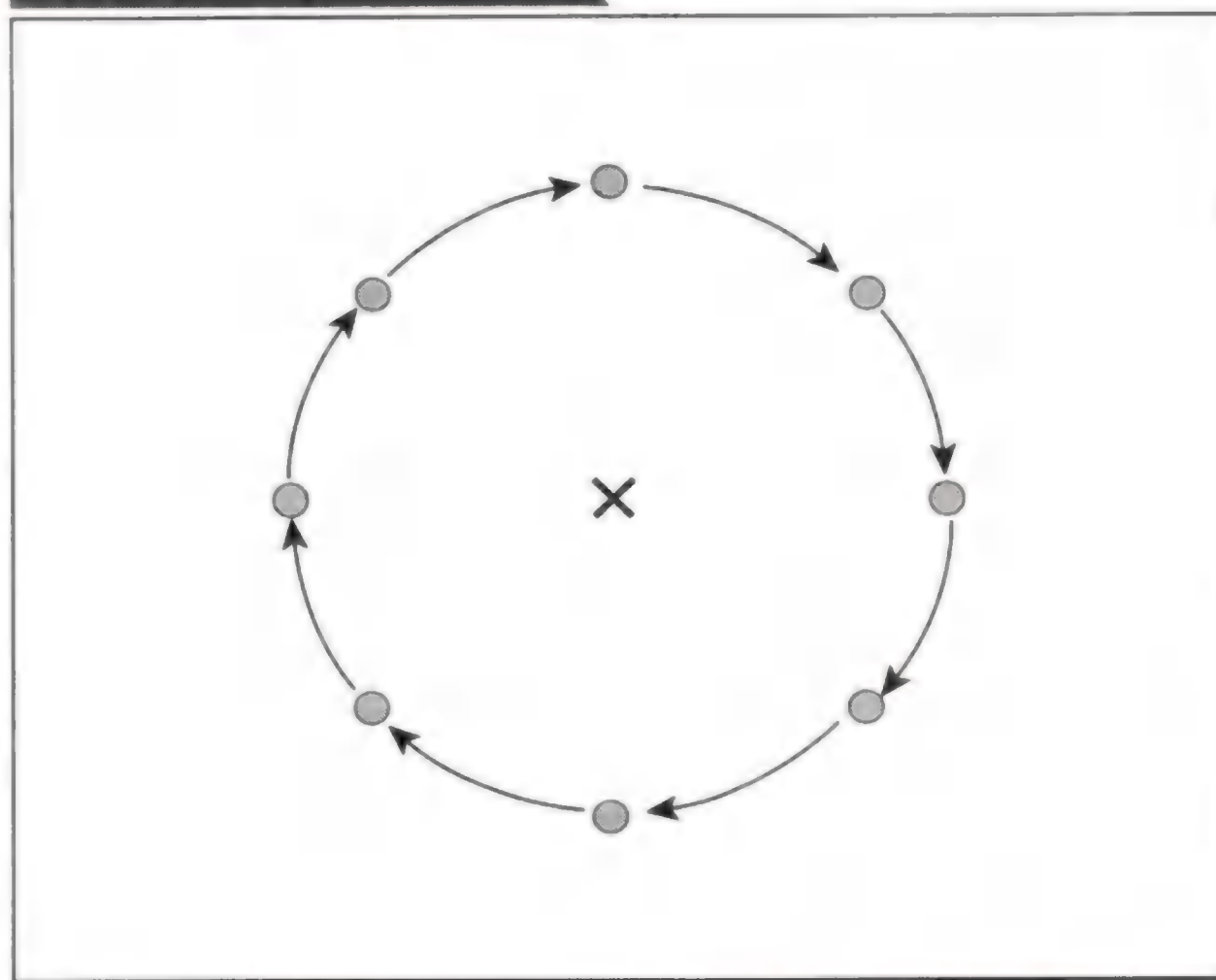



Fig. 2-62 2種類の回転弾を組み合わせたもの

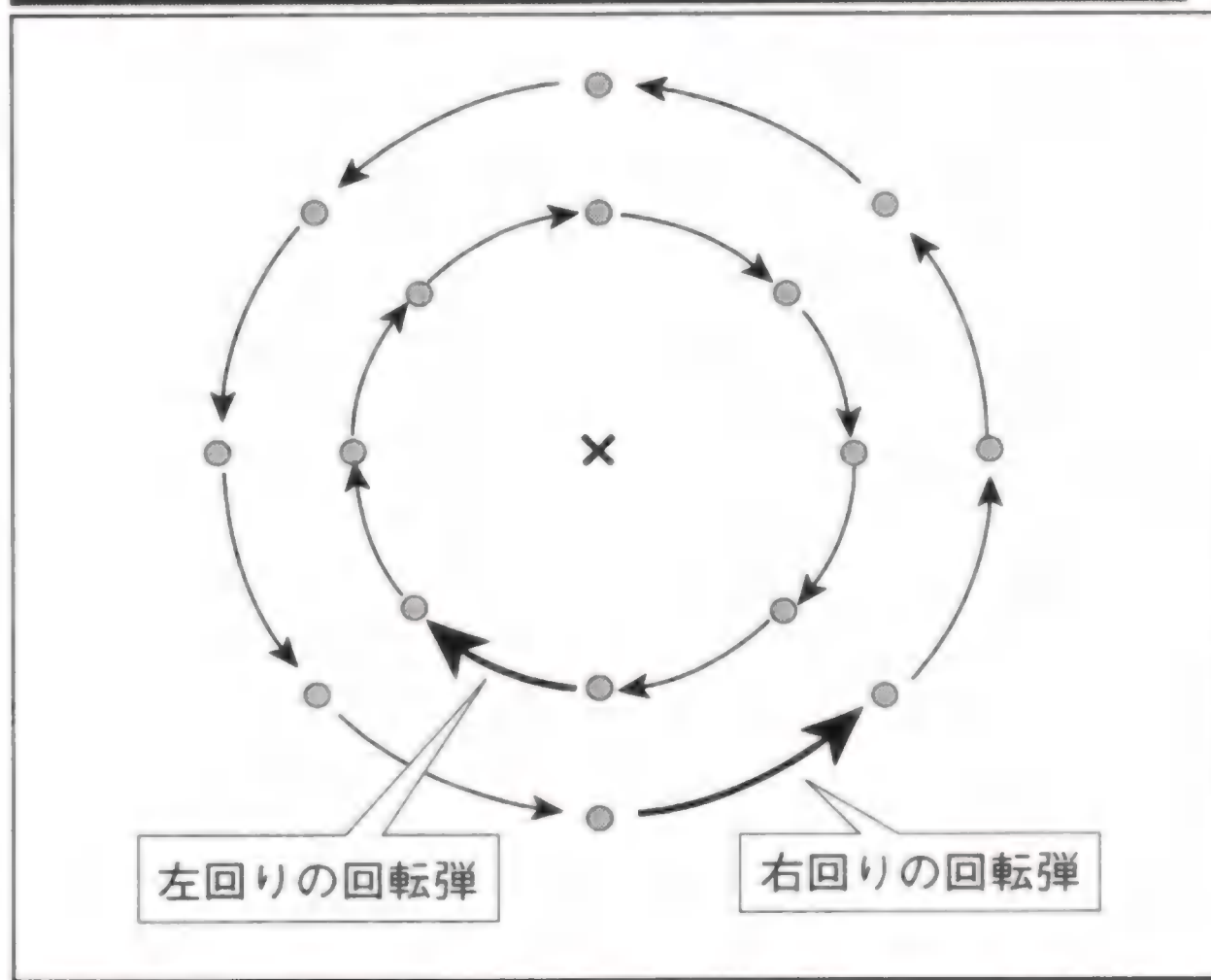



Fig. 2-63 回転弾の位置を計算する

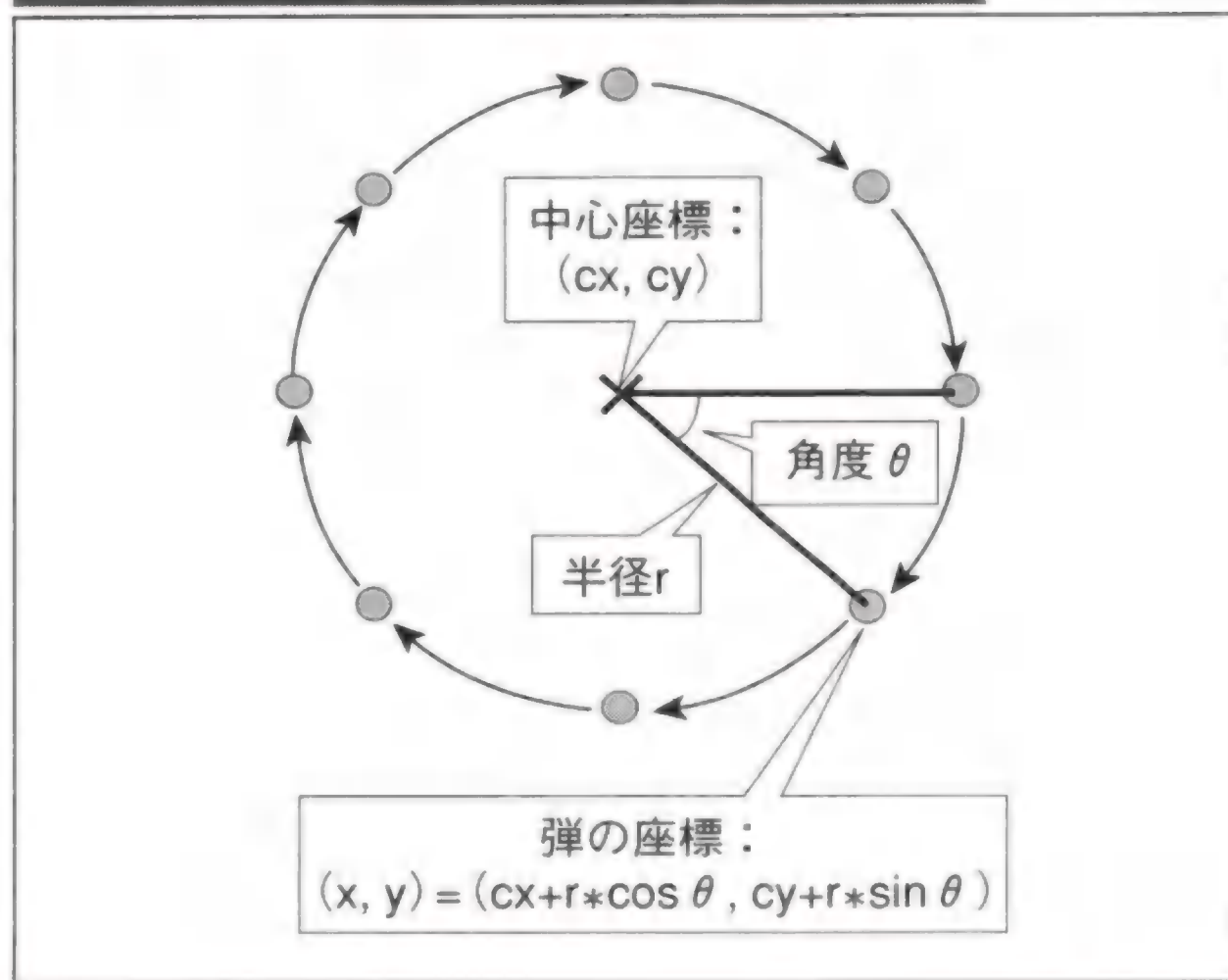
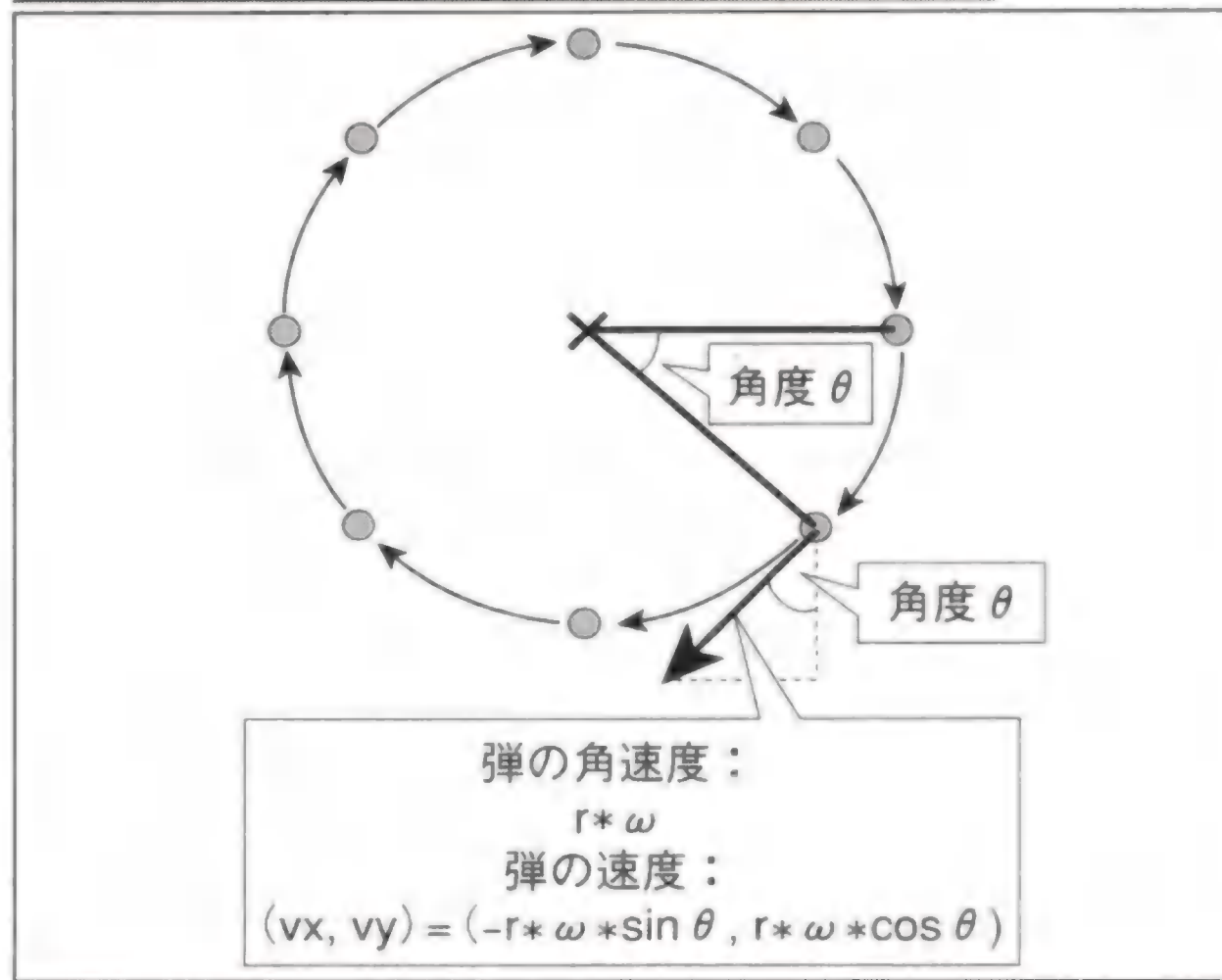
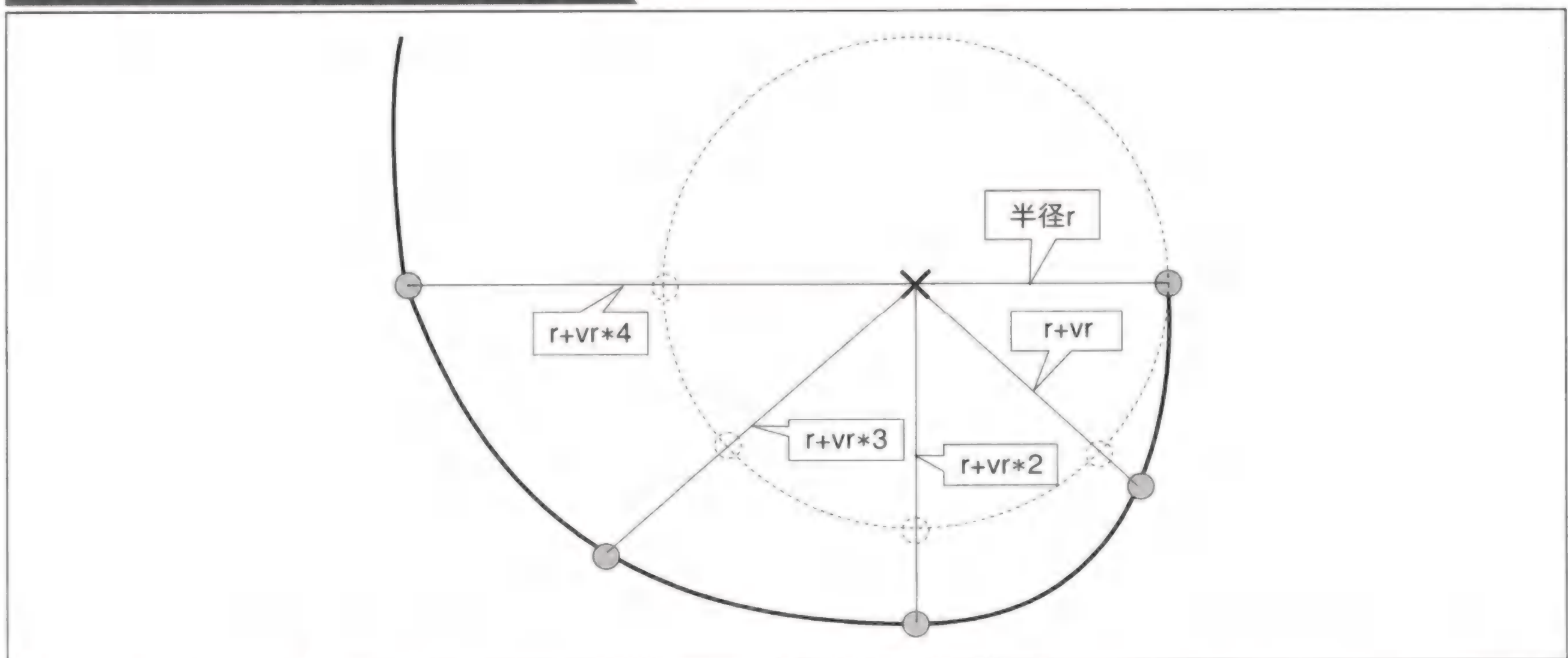



Fig. 2-64 回転弾の速度を計算する

List 2-23 回転弾の移動

```
#include <math.h>

void MoveLoopingBullet(
    float& x, float& y,     // 弾の座標
    float& vx, float& vy,   // 弾の速度
    float cx, float cy,     // 回転中心の座標
    float r,                // 半径
    float theta,            // 角度(ラジアン)
    float omega             // 1回の移動で変化する角度(ラジアン)
) {
    // 角度を変化させる
    theta+=omega;

    // 位置を計算する
    x=cx+r*cos(theta);
    y=cy+r*sin(theta);

    // 弾の速度(必要な場合だけ)
    vx=-r*omega*sin(theta);
    vy= r*omega*cos(theta);
}
```

■ 回転半径を変化させる

回転とともに回転半径を変化させると、半径が変化する回転弾を作ることができます(Fig. 2-65)。

Fig. 2-65 回転弾の半径を変化させる

最初の半径をr、半径の変化をrvとすると、1回移動したあとの半径は、

r+rv

となり、n回移動したあとの半径は、

r+rv*n

となります。単に回転するだけの弾ではもの足りないときには、このように回転半径を変化させたり、あるいは回転速度を加速度的に変化させたりすると面白さが増すでしょう。

List 2-24は半径が変化する回転弾を移動させるプログラムです。

サンプル

- 回転弾 → P. 314
- 回転弾2 → P. 314

List 2-24 半径が変化する回転弾の移動

```
#include <math.h>

void MoveLoopingBullet2(
    float& x, float& y,     // 弾の座標
    float& vx, float& vy,   // 弾の速度
    float cx, float cy,     // 回転中心の座標
    float& r,               // 半径
    float vr,               // 半径の変化
    float theta,            // 角度(ラジアン)
    float omega             // 1回の移動で変化する角度(ラジアン)
) {
    // 角度を変化させる
    theta+=omega;

    // 半径を変化させる
    r+=vr;

    // 位置を計算する
    x=cx+r*cos(theta);
    y=cy+r*sin(theta);

    // 弾の速度(必要な場合だけ)
    vx=-r*omega*sin(theta);
    vy= r*omega*cos(theta);
}
```

狙い撃ち弾+回転弾

一見複雑に見える弾の動きも、実は単純なアルゴリズムの組み合わせでできていることがよくあります。そのような弾の一例として、自機の方向に飛ぶ弾と回転弾の組み合わせを紹介します(Fig. 2-66)。

Fig. 2-66 自機の方向に飛ぶ弾+回転弾

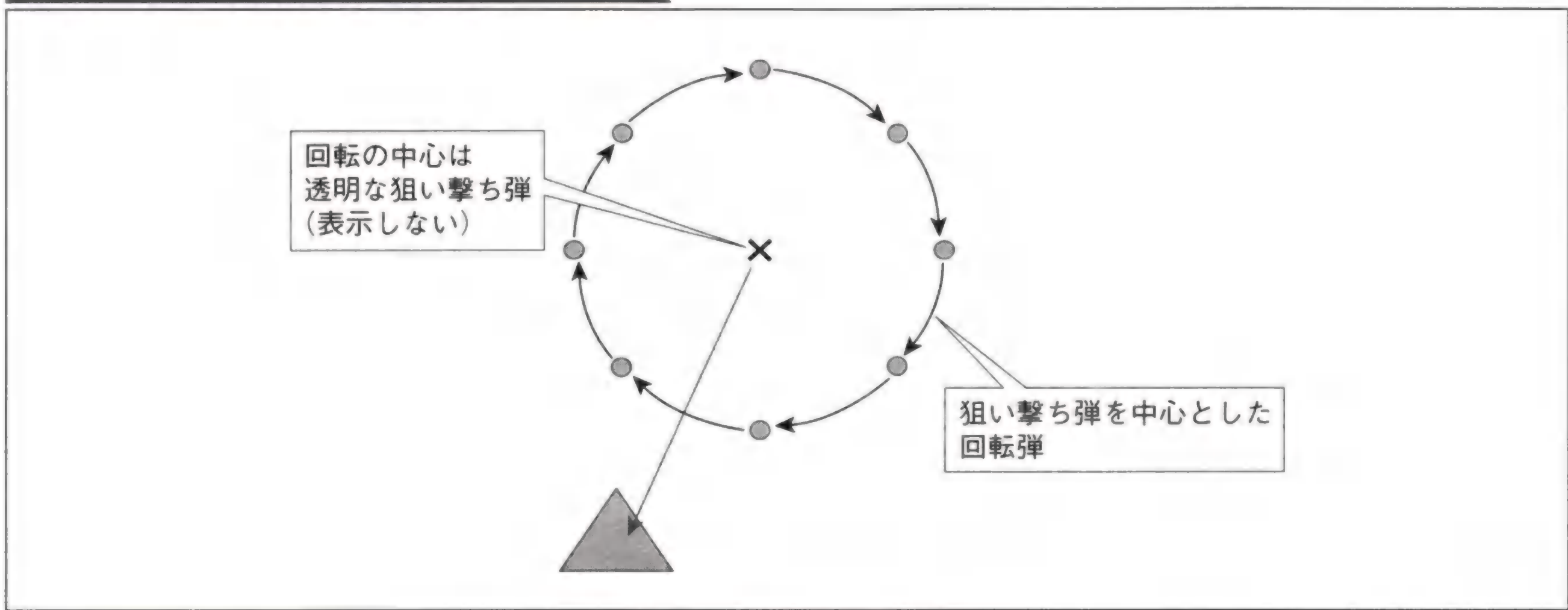



Fig. 2-66は狙い撃ち弾(→ P. 10)と回転弾(→ P. 58)の動きを組み合わせた弾です。ここでは自機の方向に飛ぶ弾は表示せずに、単に回転弾の中心として使います。このように「透明な弾」(表示はしないが座標計算はする弾)を使うと、一見しただけでは動きの仕組みがわかりにくい弾を作ることができます。

■ 回転の中心となる弾の消去

弾が画面外に出たら消す必要がありますが、回転の中心となる弾の場合には、消去するタイミングに注意が必要です。「狙い撃ち弾+回転弾」の場合、回転の中心となる透明な狙い撃ち弾が画面外に出たときに単純に消去してしまうと、回転の中心がなくなってしまうため、画面内に残った回転弾の動きを計算することができなくなります(Fig. 2-67)。画面内に回転弾が残っているかぎり、回転の中心となる弾は消してはいけないのです。

そこで、中心の弾はすべての回転弾が画面外に出たあとに消去します(Fig. 2-68)。これならば中心の弾が画面外に出たあとでも、画面内に残った回転弾の動きを正しく計算することができます。

サンプル

狙い撃ち弾+回転弾 → P. 314

Fig. 2-67 回転中心の間違った消去方法

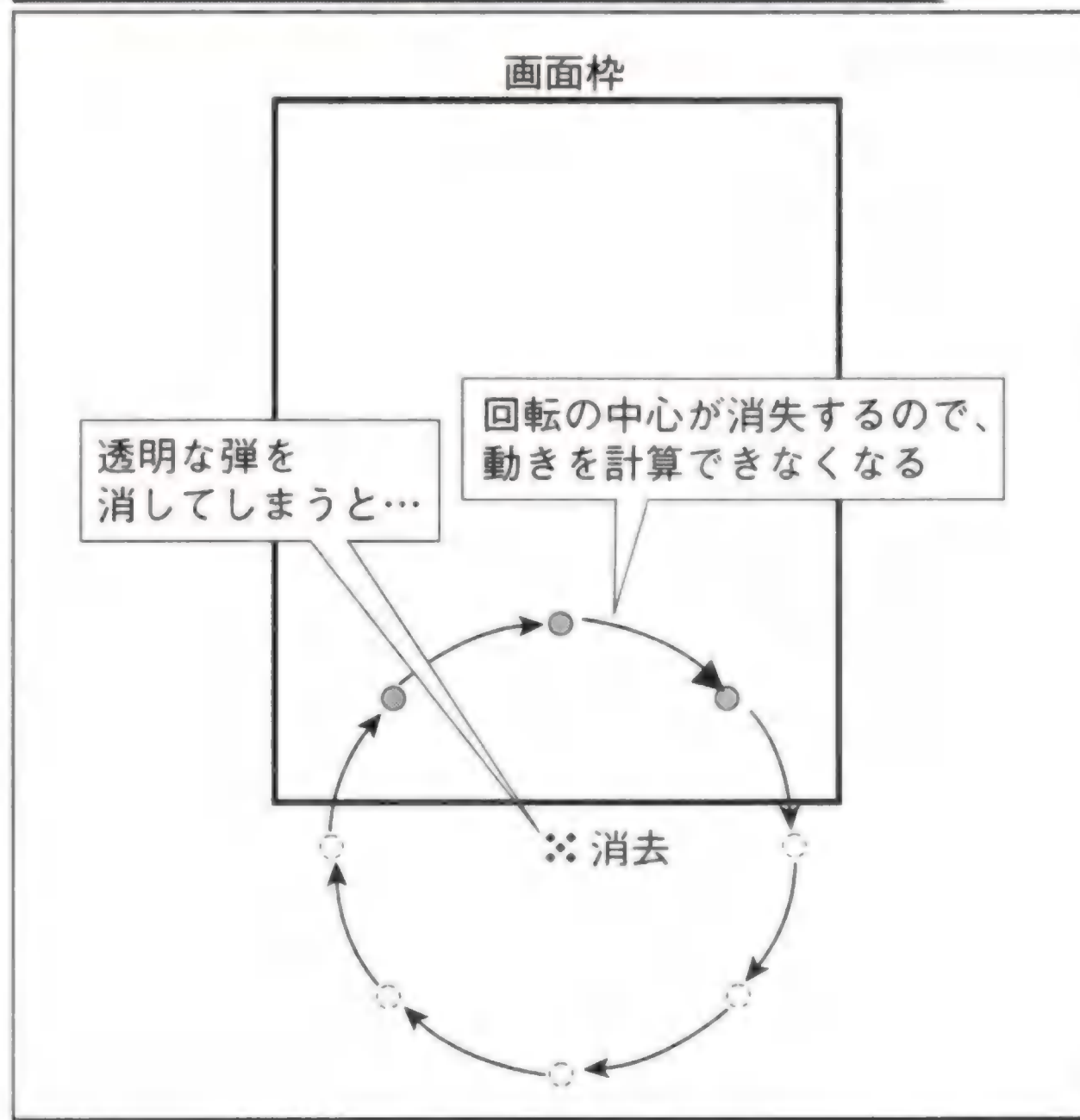
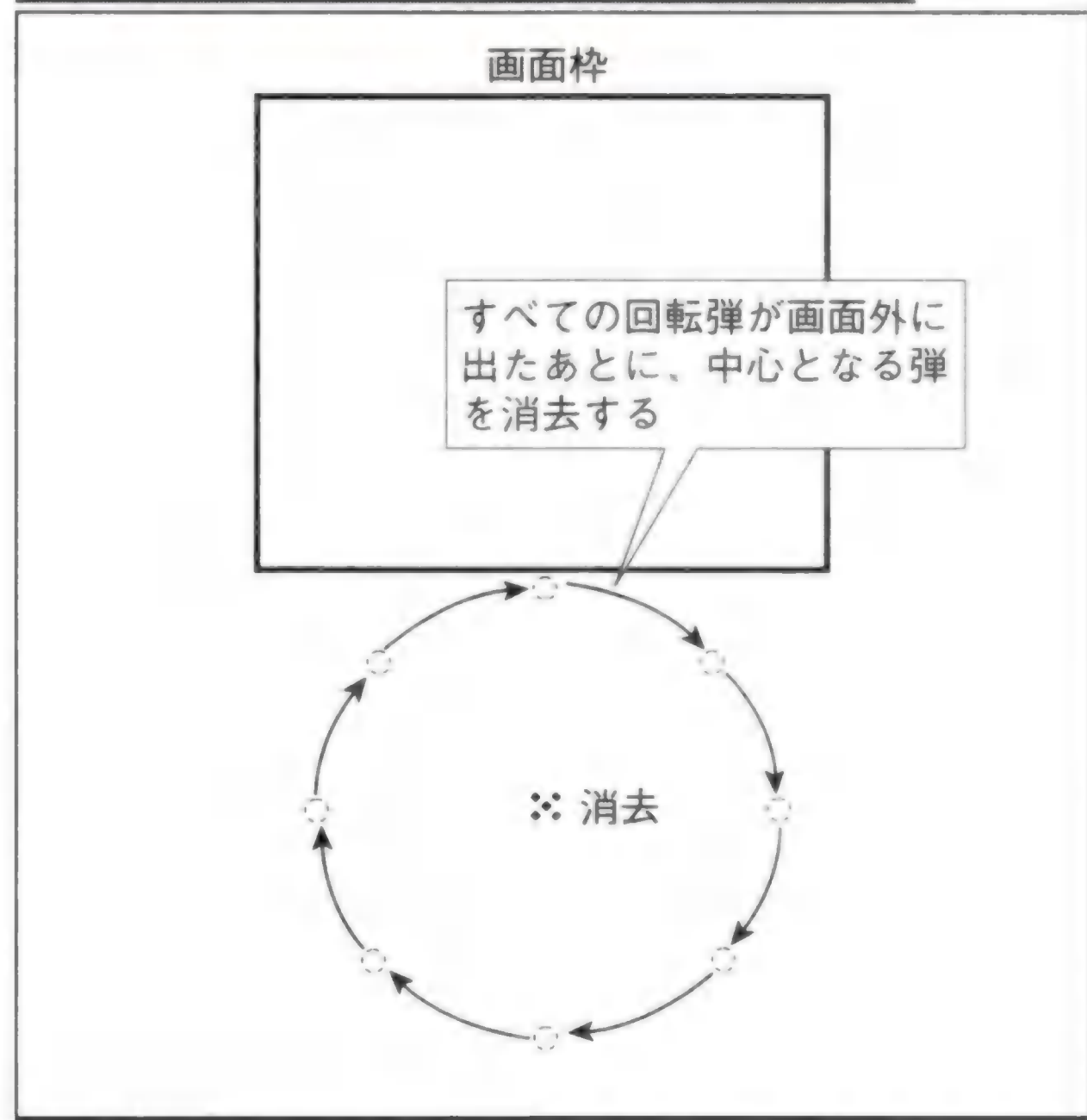
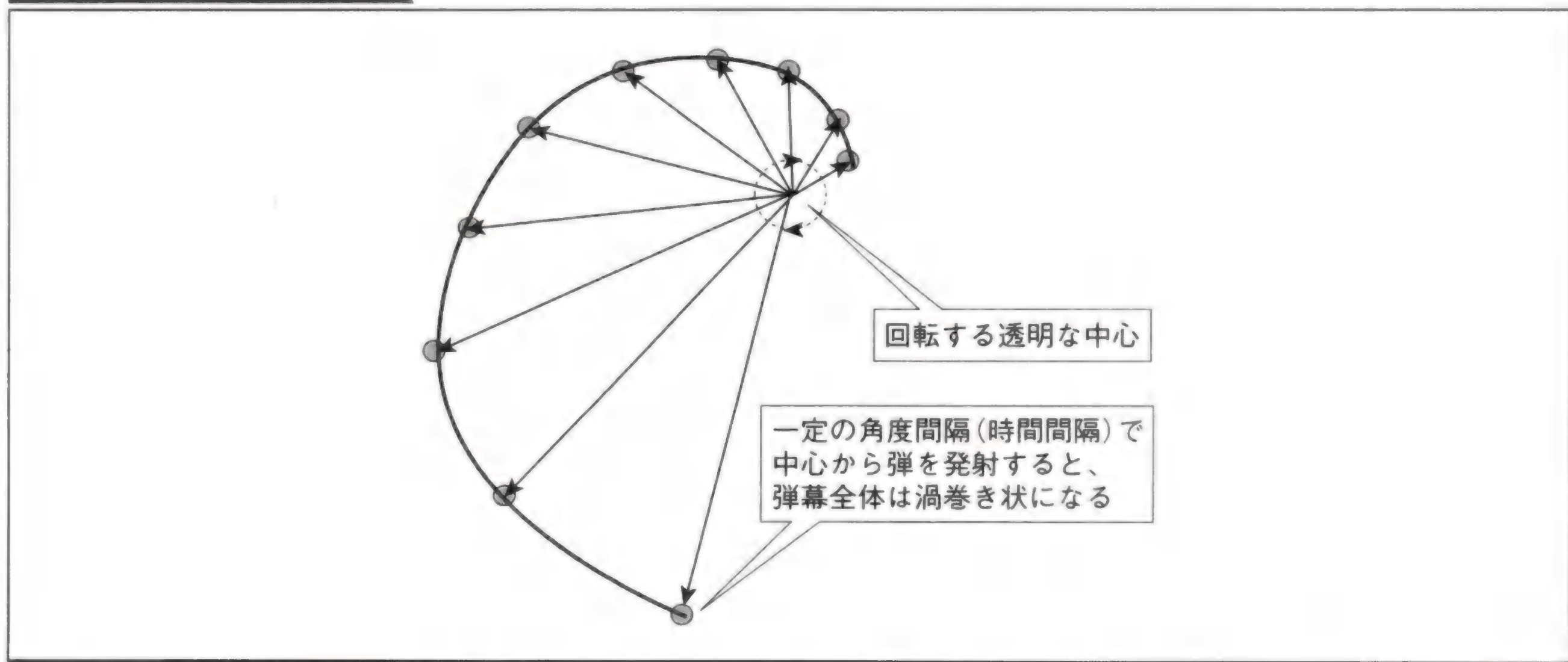



Fig. 2-68 回転中心の正しい消去方法

渦巻き弾

渦巻き状に広がる弾です。回転弾（→ P. 58）の回転半径を変化させると個々の弾の動きは渦巻き状になりますが、全体としては渦巻き状には見えず、円形の弾幕の半径が変化しているように見えます。弾幕全体の形を渦巻き状にするには、Fig. 2-69のような方法を使います。

Fig. 2-69 渦巻き弾

ポイントは回転する透明な中心です。中心を回転させておいて、一定間隔で回転方向に向かって直進する弾を発射すると、弾幕全体の形は渦巻き状になります。ある弾が発射されるときには、1つ前に発射された弾が少しずつ進んでいるため、弾幕が渦巻き状になるのです。個々の弾は単純な方向弾なのですが、このように発射の方法を工夫することによって、面白い弾幕が形作られます。

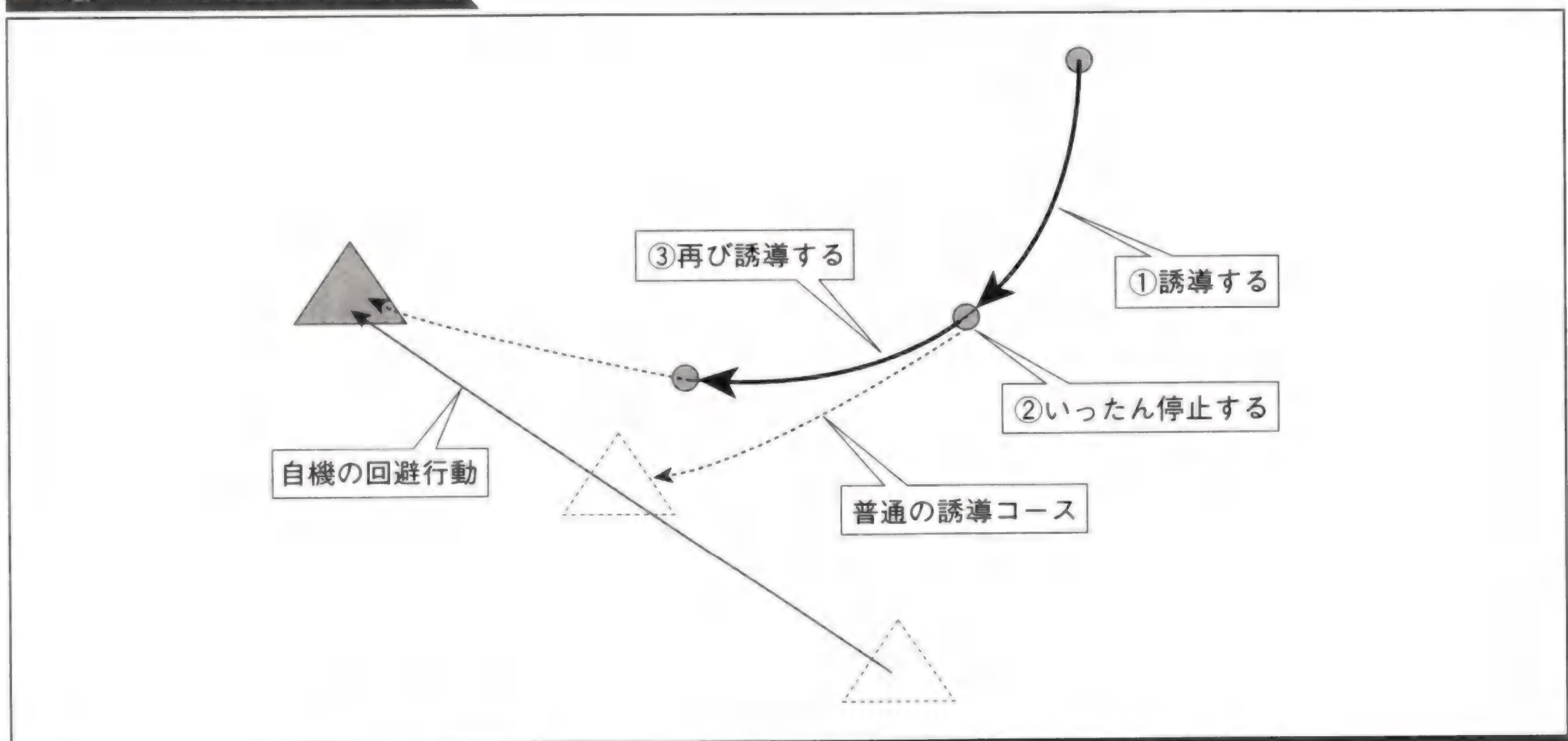
サンプル

渦巻き弾 → P. 314

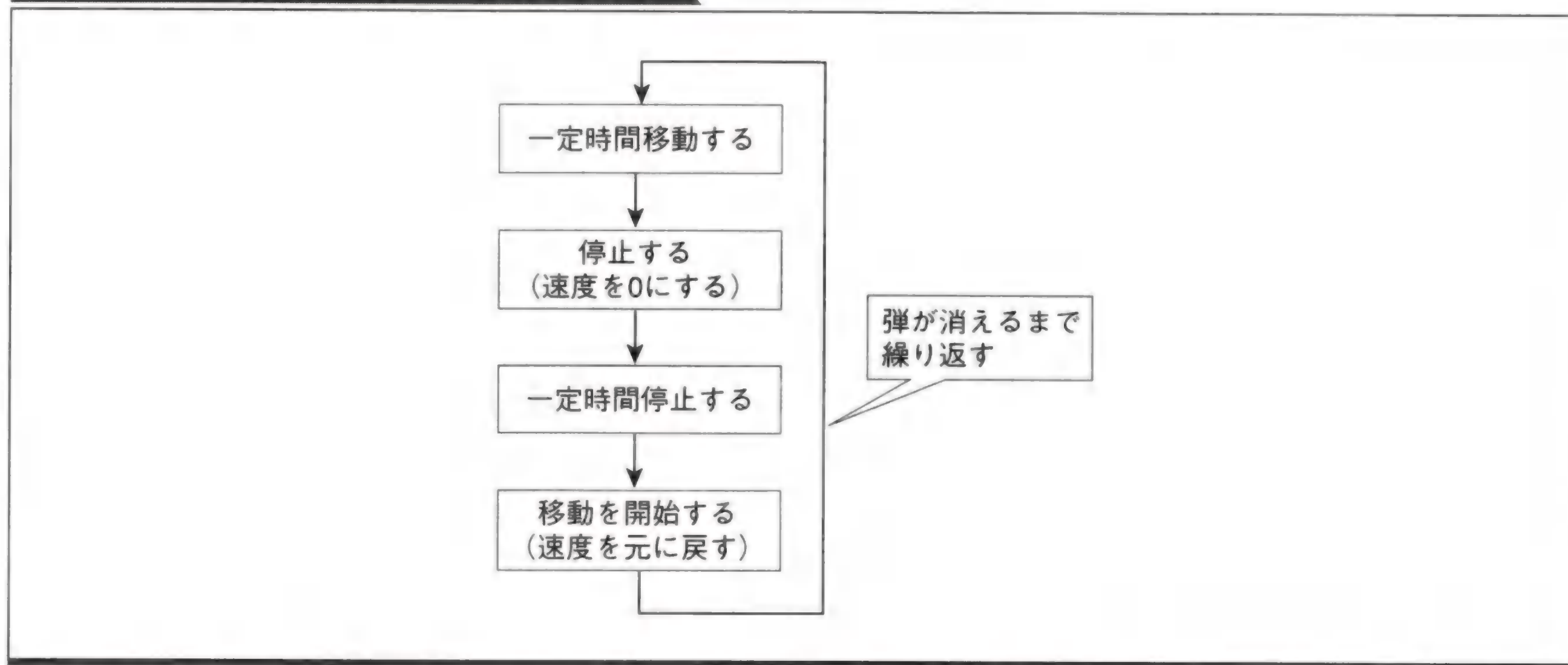
誘導弾のアレンジ

誘導弾（→ P. 39）の動きに変化をつけると、普通の誘導弾にはないアクションが生まれるとともに、見た目も楽しくなります。よけられない誘導弾をよけられるようにしたり、逆に簡単によけられるはずの誘導弾を難しくしたりすることもできます。

たとえば、動いたり止まったりする誘導弾などは簡単に作れます（Fig. 2-70）。最初は通常の誘導弾と同じように自機を追尾しますが、途中でいったん停止します。そして一定時間停止したあとに、再び自機の追尾を開始します。あとは移動と停止の繰り返しです。

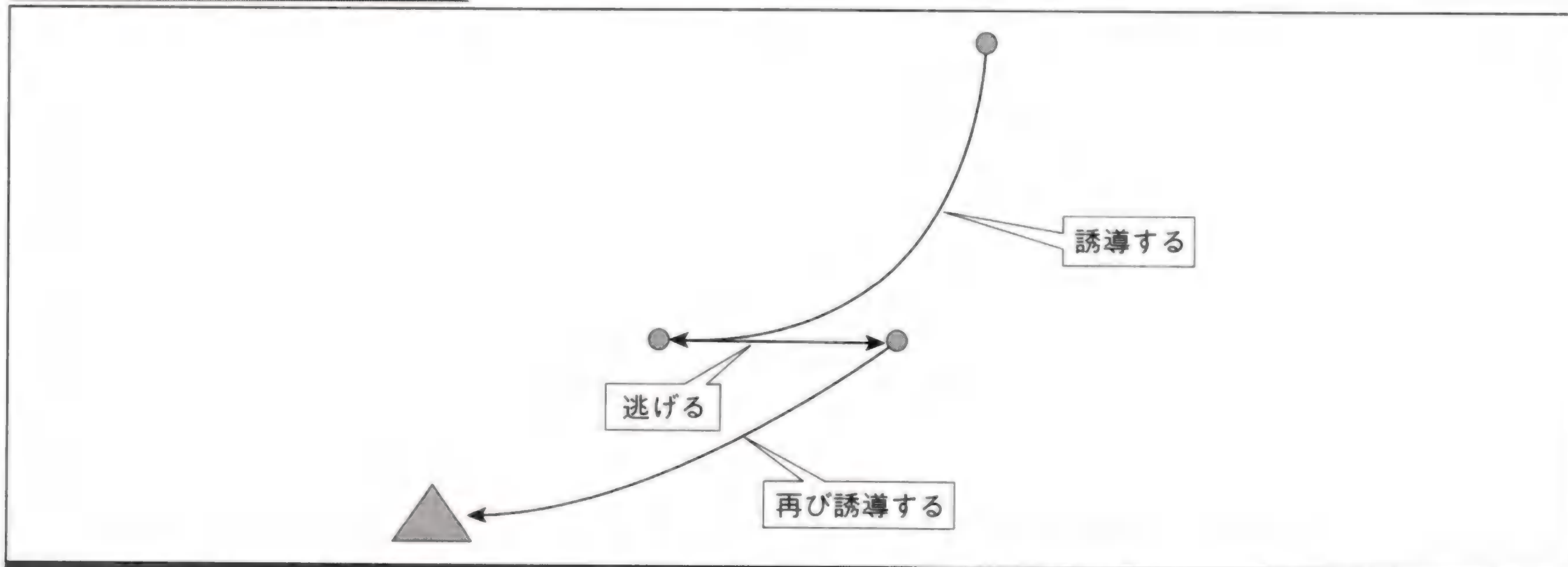
Fig. 2-70 停止する誘導弾

■ 停止する誘導弾

単純に誘導弾の速度を0に設定すれば、誘導弾を停止させることができます。停止したり移動したりさせるには、一定時間ごとに速度を0にしたり元の値に戻したりすればよいのです(Fig. 2-71)。

Fig. 2-71 停止する誘導弾のアルゴリズム

■ 逃げる誘導弾

通常の誘導弾の速度はプラスの値ですが、マイナスの値を設定するとどうなるでしょうか。この場合は、自機から遠ざかる方向に動く弾、つまり自機から逃げる弾になります。自機から逃げるだけではあまり使えませんが、自機を追尾する期間と自機から逃げる期間を一定時間ごとに切り替えるようにすると、トリッキーな動きをする弾ができます(Fig. 2-72)。

逃げる誘導弾のアルゴリズムは、停止する誘導弾のアルゴリズム(Fig. 2-71)とほぼ同じです。速度を0に設定するかわりに、マイナスに設定すればよいのです。レーザーなどの場合に

Fig. 2-72 逃げる誘導弾

は速度を急激に変化させるのではなく、加速度を使って徐々にプラスからマイナス、マイナスからプラスへと変化させたほうが、なめらかな動きになります。

誘導弾の動きを変化させると、普通の誘導弾とは回避方法が違った弾ができます。たとえば「逃げる誘導レーザー」の場合、自機からときどき逃げるぶん回避が簡単になるように思えますが、実際には普通の誘導レーザーよりもずっと回避が難しくなります。これは、単純な動きではなくなるために単純な回避ができないことと、レーザーが逃げるために画面内の滞在時間が長くなって、画面内に存在するレーザーの本数が増えることが理由です。

サンプル

- 停止する誘導弾　→ P. 314
- 逃げる誘導レーザー　→ P. 314

直進するビーム

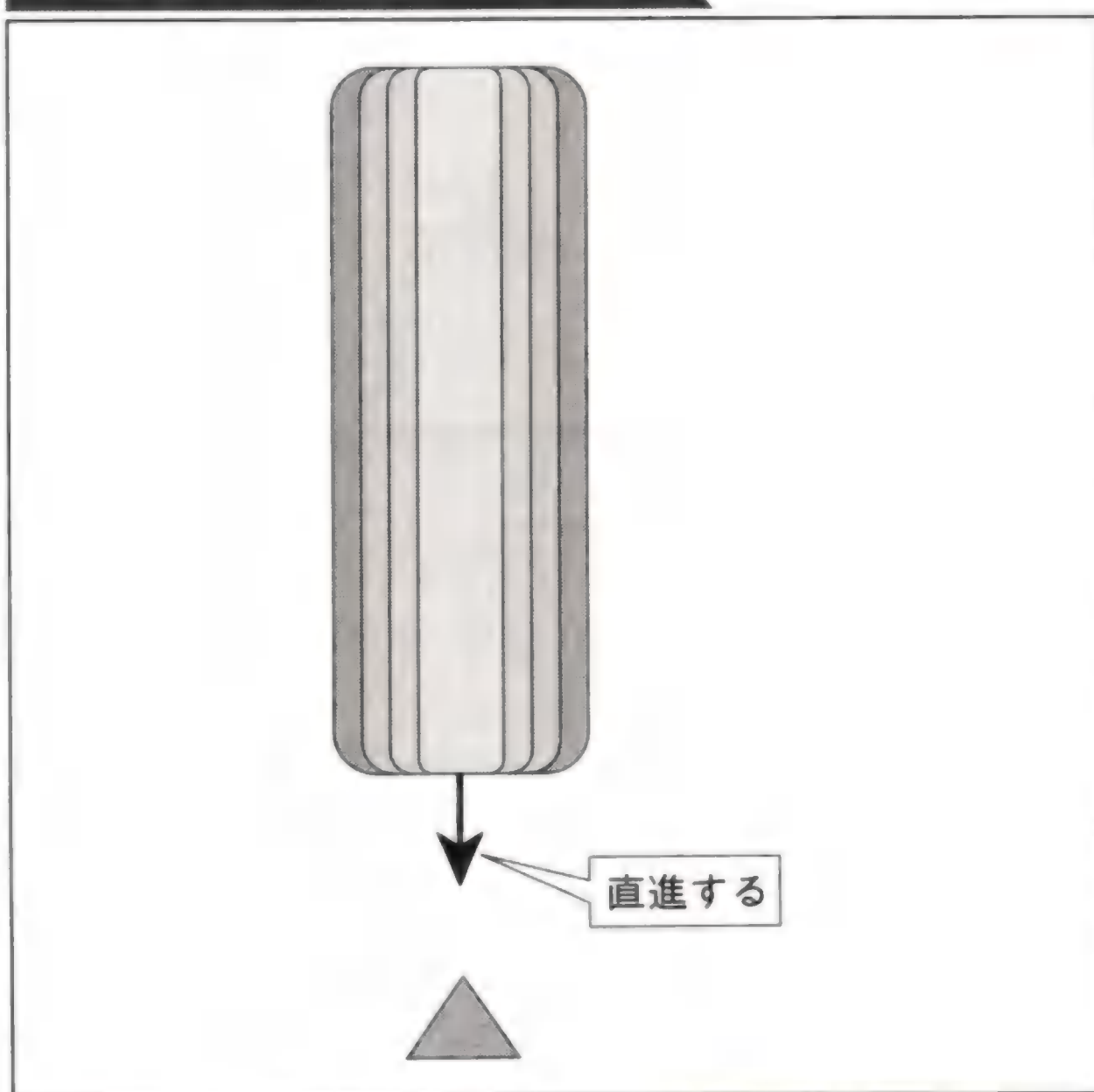
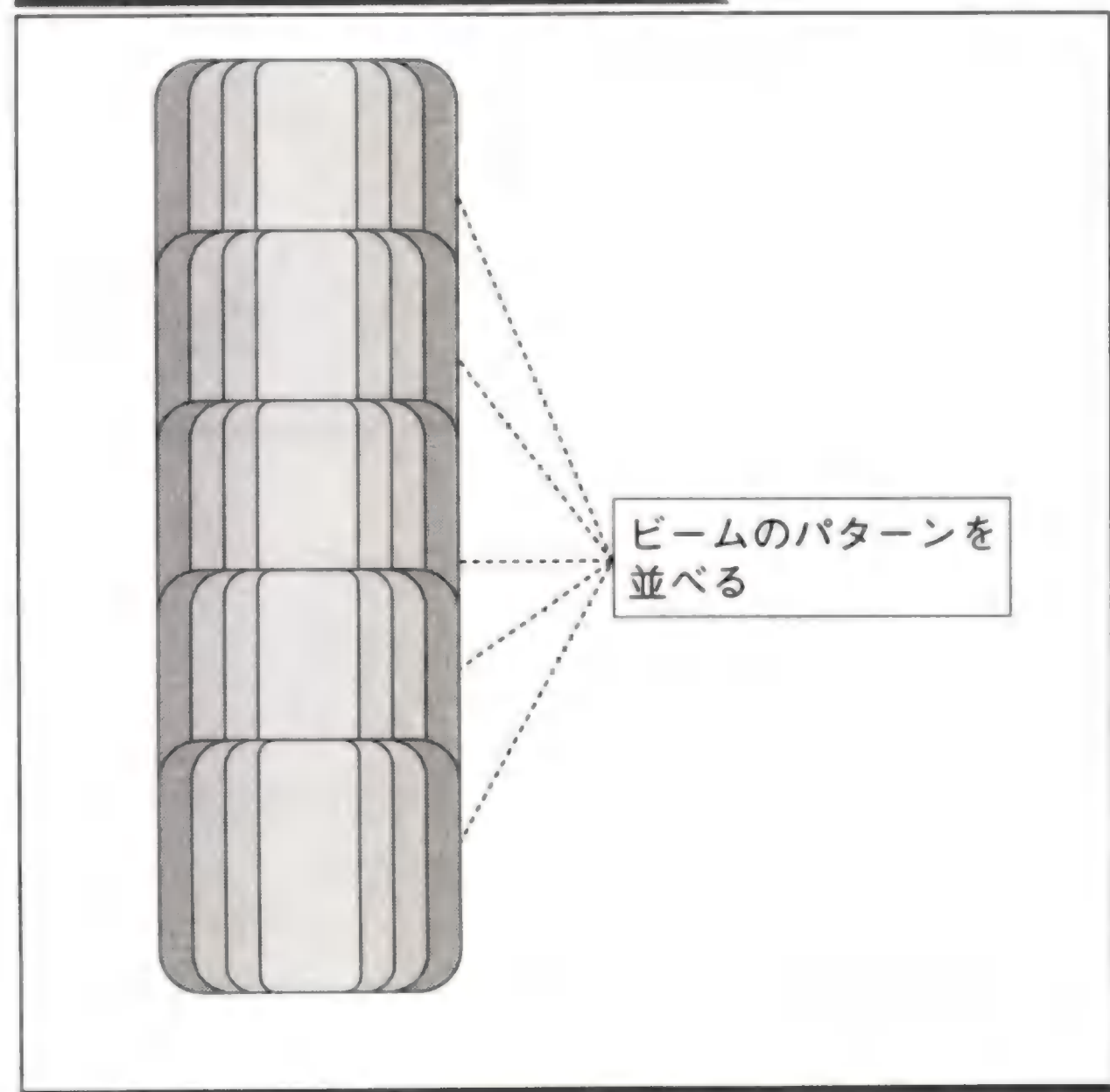
直進する太いビームです（Fig. 2-73）。最近のボスキャラは濃密な弾幕を張るものが多いですが、ひと昔前はこのような太い（そして高速の）直進ビームを撃ってくるのがメインでした。

太いビームは誘導レーザーに比べると簡単に作れます。ビームのパターンを用意しておき、これを並べて好きな長さにすればよいのです（Fig. 2-74）。ただ、太くて高速なビームはよけにくいので、発射前には、「これからビームが発射される」という攻撃の予兆をプレイヤーに示す工夫が必要となります。

Fig. 2-73 直進するビーム

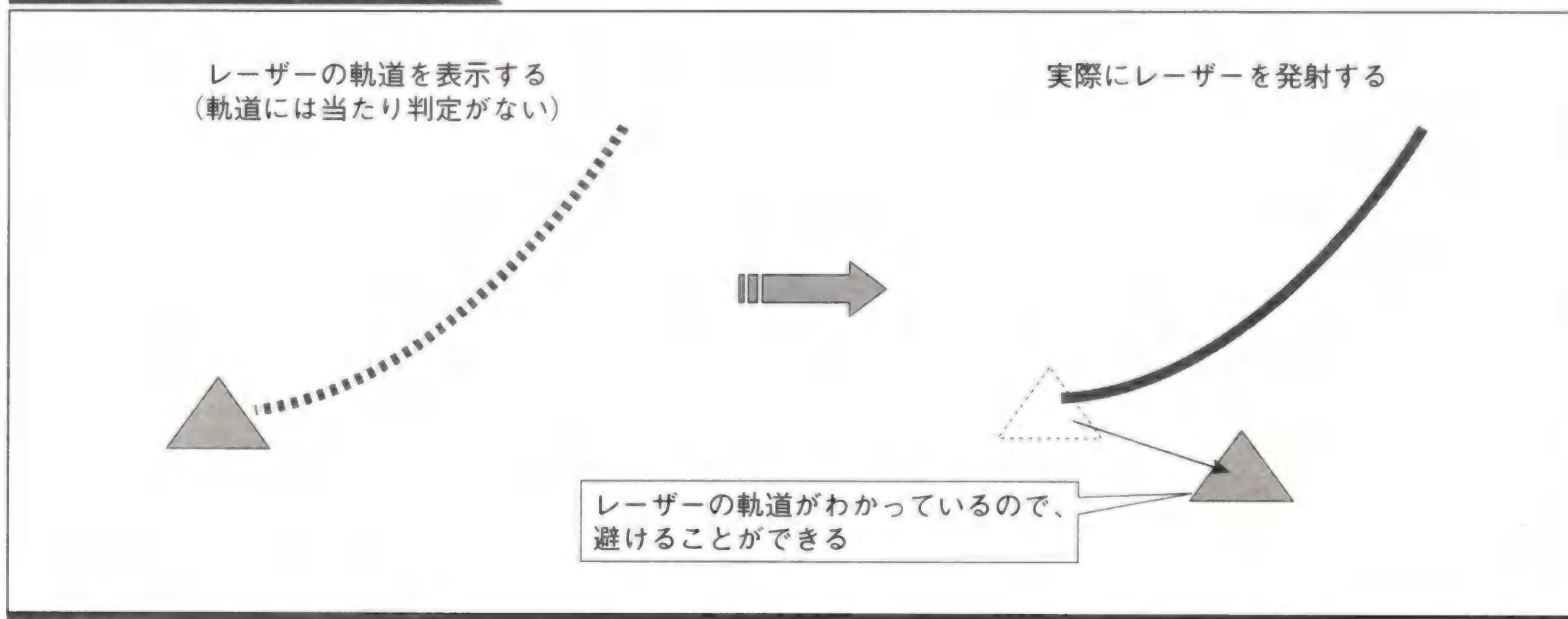
Fig. 2-74 ビームの作り方

サンプル

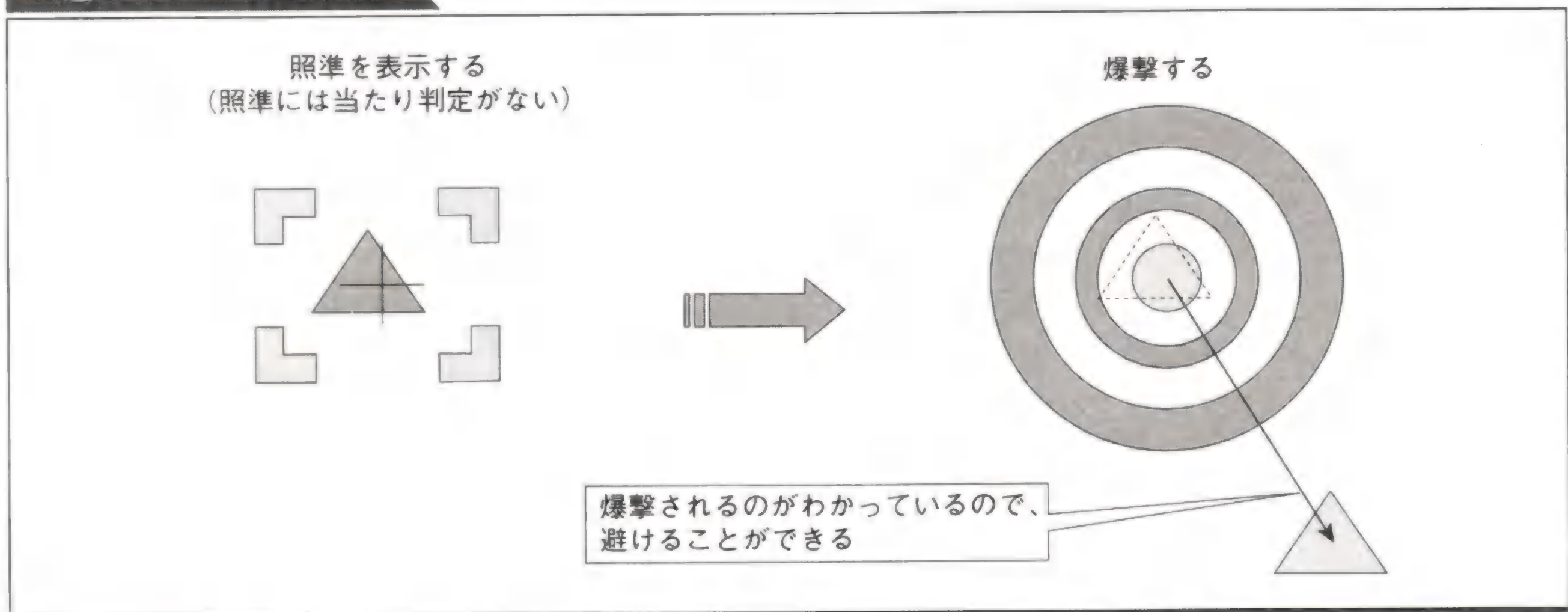
直進するビーム →P. 314

攻撃の予兆

高速なレーザーや太いビームが何の前触れもなしに飛んできて撃墜されたら、プレイヤーは「悔しい」というよりも「このゲームは理不尽だ」と感じてしまうでしょう。こういった奇襲は現実の攻撃手段としては有効かもしれませんが、シューティングゲームの目的は「プレイヤーを撃墜すること」ではなく「プレイヤーに適度なスリルを味わわせて楽しませること」だということを忘れてはいけません。そこで、本来の「奇襲」の意味からは外れますが、「これから奇襲しますよ」ということを何らかの方法でプレイヤーに伝えておく必要があります。

たとえば高速なレーザーの場合には、レーザーが実際に発射される前にレーザーの軌道を表示するのが1つの方法です(Fig. 2-75)。この軌道には当たり判定はありません。プレイヤーは軌道を見てレーザーが間もなく発射されることを知り、回避行動をとることができます。

Fig. 2-75 レーザーの予兆

また、広範囲への爆撃などもとっさの回避が難しいので、プレイヤーに対してあらかじめ何らかの警告をしておくのがよいでしょう。たとえば、照準を出してから爆撃するなどの方法があります(Fig. 2-76)。

このほかにも、たとえば「ボスがビームを撃つ前には砲台が光る」とか、「レーザーが飛んでくる前にはエネルギーを溜めているような音がする」などといった警告の方法があります。

Fig. 2-76 爆撃の予兆

安全地帯とその対策

安全地帯というのは、その場所に自機を一度移動させれば、あとは自機をまったく動かさなくてもすべての弾がよけられる場所のことです。「安全地帯」を略して「安地(あんち)」と呼ぶこともあります。シューティングゲーマーは効率的なプレイのために安全地帯を探しますが、逆にゲームの作者は不用意に安全地帯を残さないように気を配らなければなりません(意図的に安全地帯を残すことはあります)。

たとえばFig. 2-77のようなn-way弾(→ P. 33)の場合、n-wayの扇形の外側と、弾と弾との間は安全地帯になります。ここに自機を入れてしまえば、あとは自機を動かさなくても弾には当たりません。

Fig. 2-77よりも弾を増やした場合でも、狭くはなりますが、やはり安全地帯は残ります(Fig. 2-78)。n-way弾の軌道は固定されているので、このように弾を増やしても安全地帯が残りがちです。

安全地帯をなくす1つの方法は、異なる種類の弾を混ぜることです。たとえば、n-way弾に狙い撃ち弾(→ P. 10)を混ぜれば、安全地帯はなくなります(Fig. 2-79)。狙い撃ち弾は発射された瞬間の自機の位置を狙うので、自機が動かなければ必ず命中します。したがって、自機はずっと止まっていることができなくなり、安全地帯はなくなるのです。

Fig. 2-77 n-way弾と安全地帯

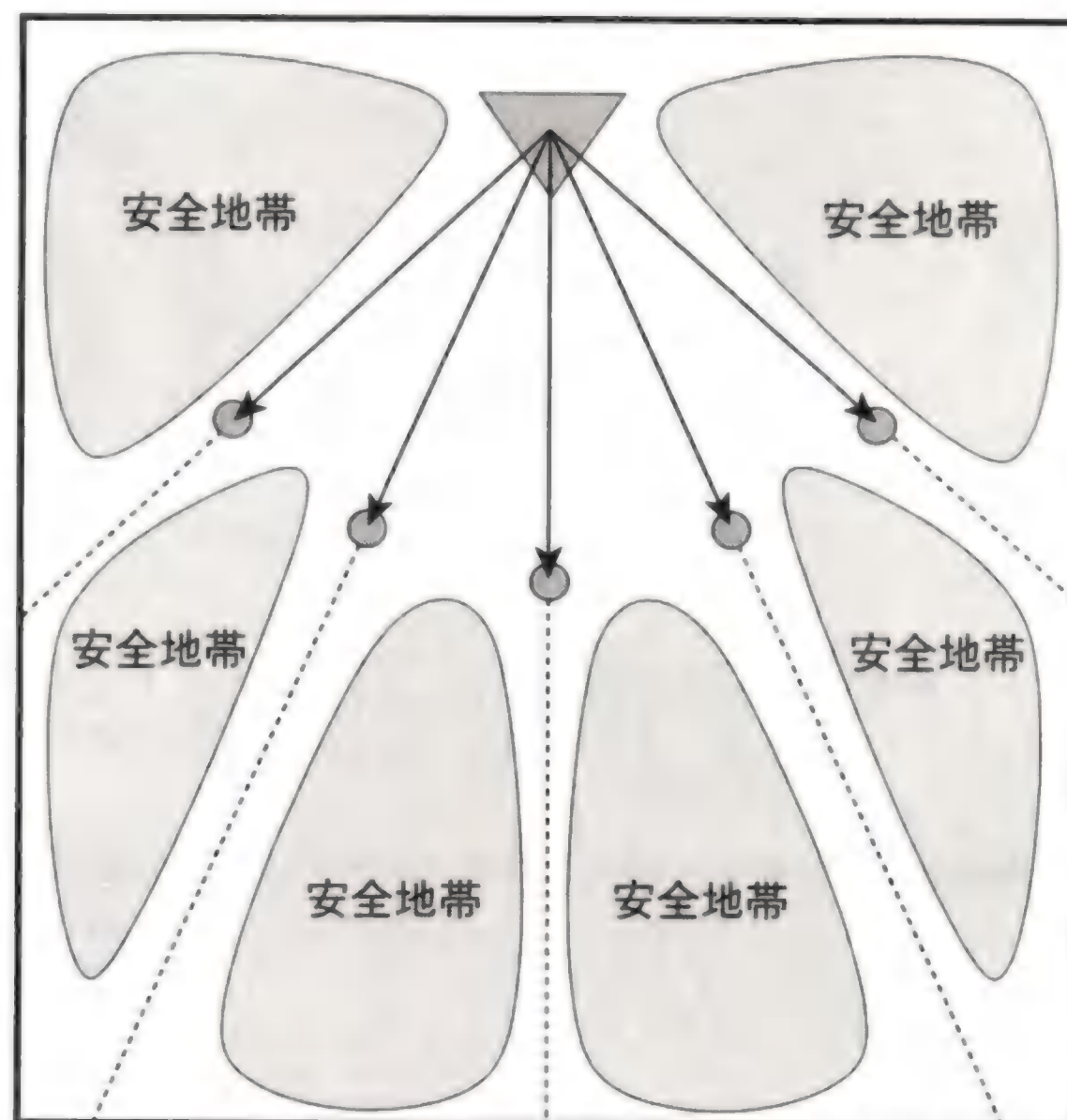



Fig. 2-78 弾を増やした場合の安全地帯

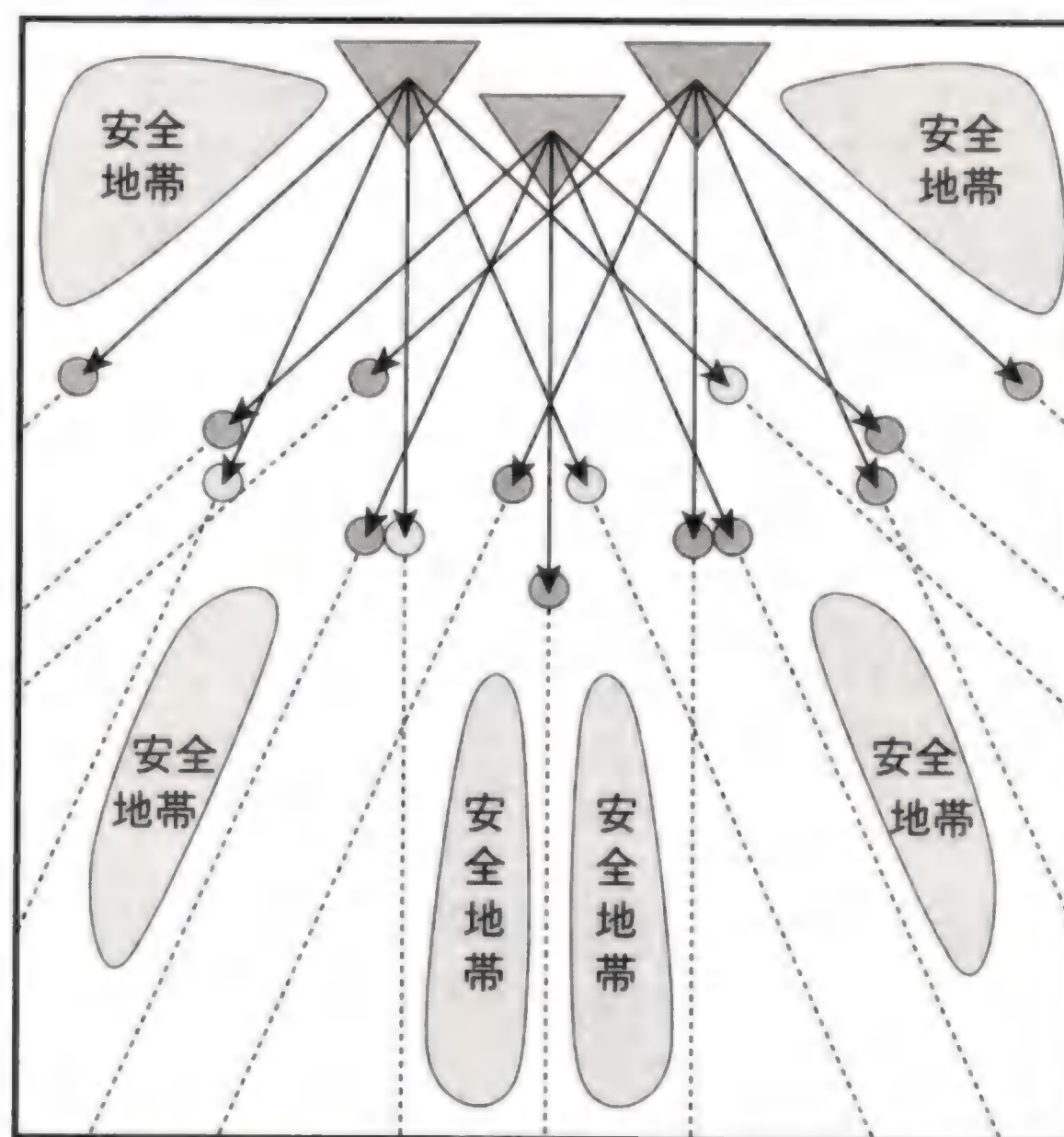
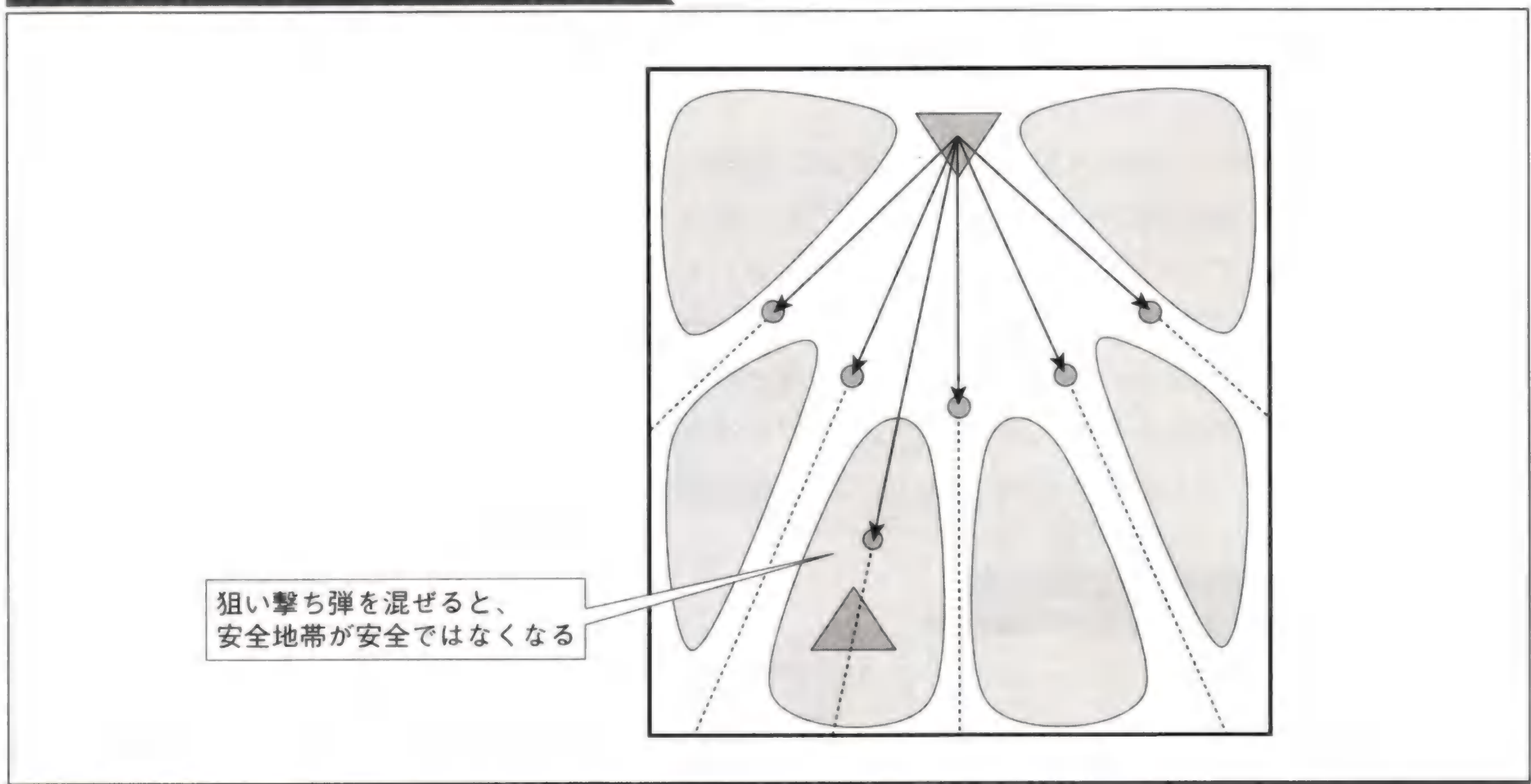



Fig. 2-79 狙い撃ち弾をn-way弾に混ぜる

狙い撃ち弾を混ぜると、
安全地帯が安全ではなくなる

n-way弾と狙い撃ち弾のように、異なる軌道の弾を組み合わせるときには、可能ならば弾の色を別々にするとプレイしやすくなります。たとえばn-way弾は青、狙い撃ち弾は赤といったように色を変えると、プレイヤーに対して「異なる軌道の弾が飛んできますよ」というメッセージを伝えることができ、ストレスがたまりにくいゲームにすることができます。

サンプル

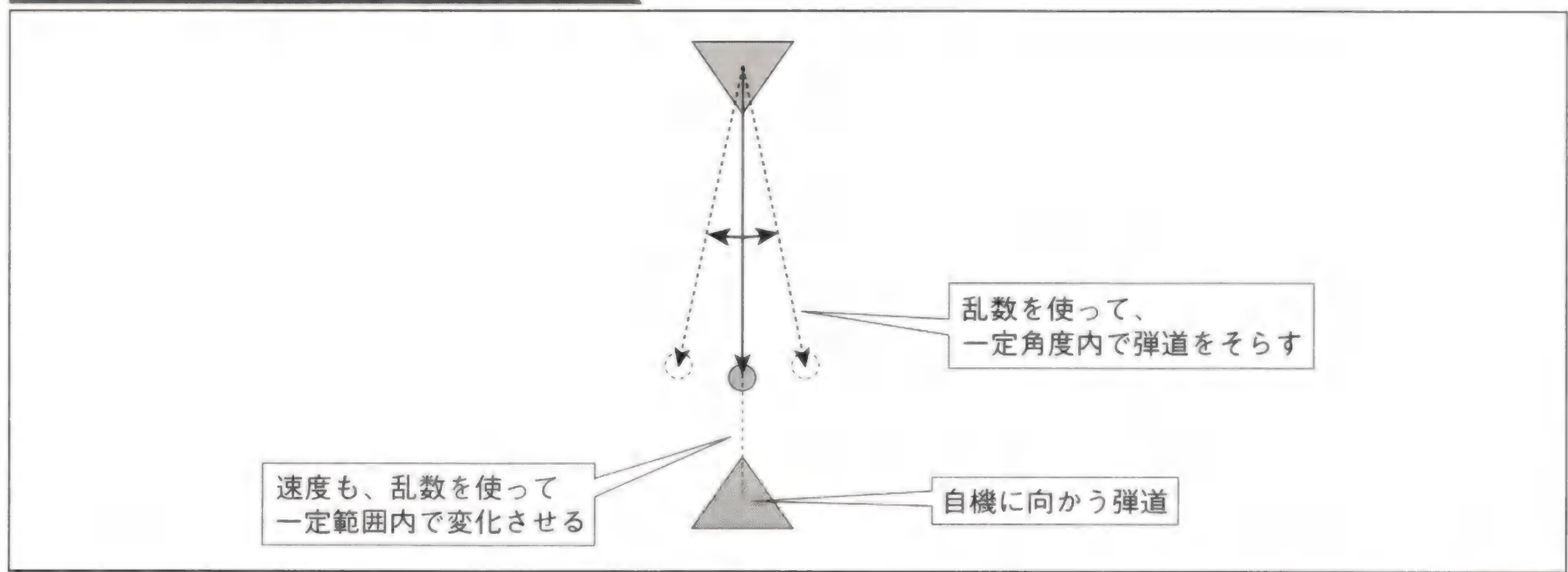
- 安全地帯 → P. 314
- 安全地帯2 → P. 314

弾の動きとランダム性

規則的な動きをする弾だけで作った弾幕は、規則的な動きだけでよけることができてしまいがちです。たとえ安全地帯をなくしたとしても、あまりに規則的な動きだけでよけられてしまう弾ばかりではつまらないでしょう。そんなときには乱数を使って、弾の動きに少しだけランダム性を加えると、刺激のある弾幕を作ることができます。

たとえば狙い撃ち弾（→ P. 10）の場合、自機に向かってまっすぐ飛ぶ弾道が基本ですが、乱数を使ってわざと弾道をずらしてやることもできます（Fig. 2-80）。自機に向かう弾道を中心線として、その周りに−10°から+10°などの一定角度内で弾道をずらします。

Fig. 2-80 角度をずらした狙い撃ち弾

乱数を使って、
一定角度内で弾道をそらす
速度も、乱数を使って
一定範囲内で変化させる
自機に向かう弾道

中心線の方向をangle0、弾道をずらす範囲をangle_rndとすると、ずらしたあとの弾道の方向angleは次の式で求められます。

angle=angle0+angle_rnd*(Rand()-0.5f)

ここではRandを0〜1までの値を生成する関数とします。C/C++の標準関数のrandを使う場合、rand関数は0からRAND_MAXまでの整数値を発生するので、これを次のように0〜1までの値に変換します。

(float)rand()/RAND_MAX

まとめると、弾道の方向は次の式で求められます。

angle=angle0+angle_rnd*((float)rand()/RAND_MAX-0.5f)

この式を使うと、angleはangle0-angle_rnd/2からangle0+angle_rnd/2までのランダムな値になります。弾道は中心線となる弾道の周りにランダムに分布します。

弾道の方向だけでなく、弾の速さをランダムに（乱数を使って）変化させるのも効果的です。たとえば平均の速さをspeed0として、速さが変化する範囲をspeed_rndとすると、変化した速さspeedは次の式で求められます。

```
speed=speed0+speed_rnd*((float)rand()/RAND_MAX-0.5f)
```

ほかの種類の弾でも、同じように角度、速度、加速度などをランダムに変化させることができます。ランダム性を加えると、弾によっては元の弾とは別物といってもいい弾幕ができることもあります。乱数を使うときのコツは、完全にランダムな動きにしてしまうのではなく、微妙なさじ加減でランダム性を入れることです。

サンプル

- 乱数 →P. 315
- 乱数2 →P. 315

Stage 2のまとめ ▶▶

「シューティングゲームを作ろう！」と思い立って、もし弾幕が嫌いでなかったら、まずは弾のプログラミングから始めることをお勧めします。というのは、いざシューティングゲームを作ろうと思って、「凝った演出」や「巨大なボスキャラ」や「派手でカッコいい武器」を作ったとしても、それだけではゲームとして成立しにくいからです。

演出やボスキャラや武器が何もなくても、弾さえあれば何とかゲームになります（さすがに自機は必要ですが）。絵が描けなくても何も問題はありません。「弾の絵」と「自機の絵」くらいなら、絵を描くのが好きな友達に頼めば描いてくれるでしょう。自分で描くのも友達に頼むのもめんどうならば、弾は●（マル）、自機は▲（サンカク）というのでもかまわないでしょう。実際、フリーソフトウェアではそういったゲームも見かけますし、ゲーム性が練れているものは絵が●や▲でも面白いものです。商用で絵が●や▲だけのゲームが一般受けするかどうかはまた別の話ですが………。

というわけで、「シューティングを作るときには、まず弾から作れ！」というのがStage 2のまとめです。

Stage 03 ▶▶

自機

MyShip

自機はシューティングゲームにおけるプレイヤーの分身です。もっともポピュラーな自機は戦闘機ですが、ゲームによってはロボットだったり潜水艦だったり、あるいは人間だったり動物だったりすることもあります。ゲームの世界観が変われば、自機の姿も変わります。

自機の種類はさまざまですが、操作の基本はどのゲームでもだいたい同じで、「スティックで自機を動かし、ボタンで弾を撃つ」というものです。説明書を読まなくてもとりあえず遊べるシンプルさが、シューティングゲームというジャンルの魅力でしょう。

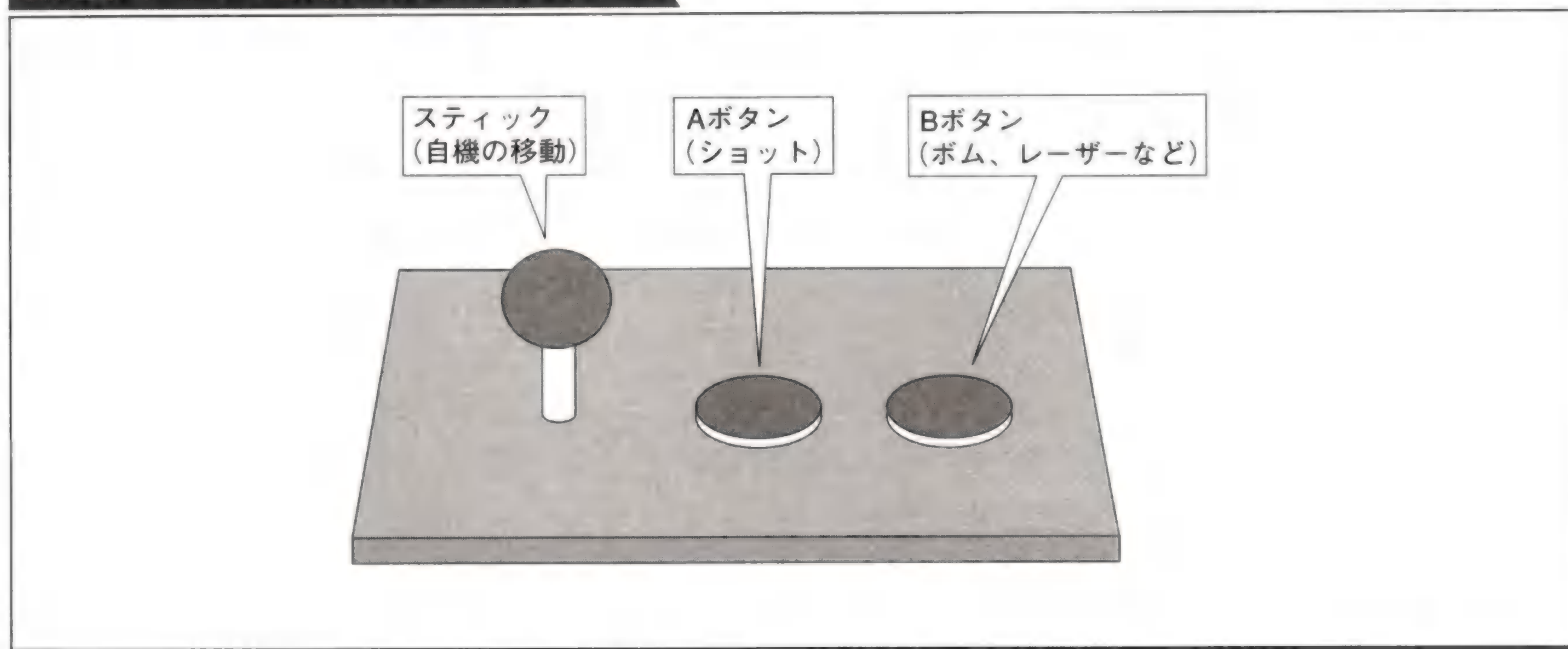
自機に関連する特殊なルール、たとえばオプションやバリアといったものはゲームによって千差万別です。ゲームによっては合体したり変形したりする自機もあります。本章ではまず基本的な自機の動きについて解説し、それからさまざまな特殊ルールについて解説していきます。

自機の移動

ほとんどのシューティングゲームは、1本のスティックと2個のボタンで操作します(Fig. 3-1)。スティックは自機の移動に使い、Aボタンはショット、Bボタンはボムやレーザーに使うのが典型的な配置です。家庭用ゲーム機やPCでは、スティックのかわりにゲームパッドやキーボードを使うことがありますが、操作の基本形は変わりません。

3個以上のボタンを使うゲームもありますが、伝統的にシューティングゲームでは「1スティック＋2ボタン」の構成が多く採用されます。古いビデオゲームの規格では使えるボタンの数が実際に少なかったためなのでしょうが、ハードウェア上の制約が少なくなった現在でもこの構成が採用されているのは、シンプルな操作系に対するこだわりでしょう。いたずらにボタンを多くしてゲームを複雑にするよりも、「1スティック＋2～3ボタン」という基本を踏襲しつつ、奥深いゲーム性を作り上げるほうが方法論として美しく思われます。

自機の移動方向は8方向が圧倒的に多いのですが、2方向（左右のみ）や4方向（上下左右のみ）のゲームもあります。スティックはデジタルスティックがほとんどで、3Dゲームを除けばアナログスティックを使うものは少ないようです。プログラミングの観点からいえば、とりあえず8方向の移動処理が作れれば、2方向や4方向は同じ要領で作れます。

Fig. 3-1 シューティングゲームの操作系

■ 8方向に自機を移動させる

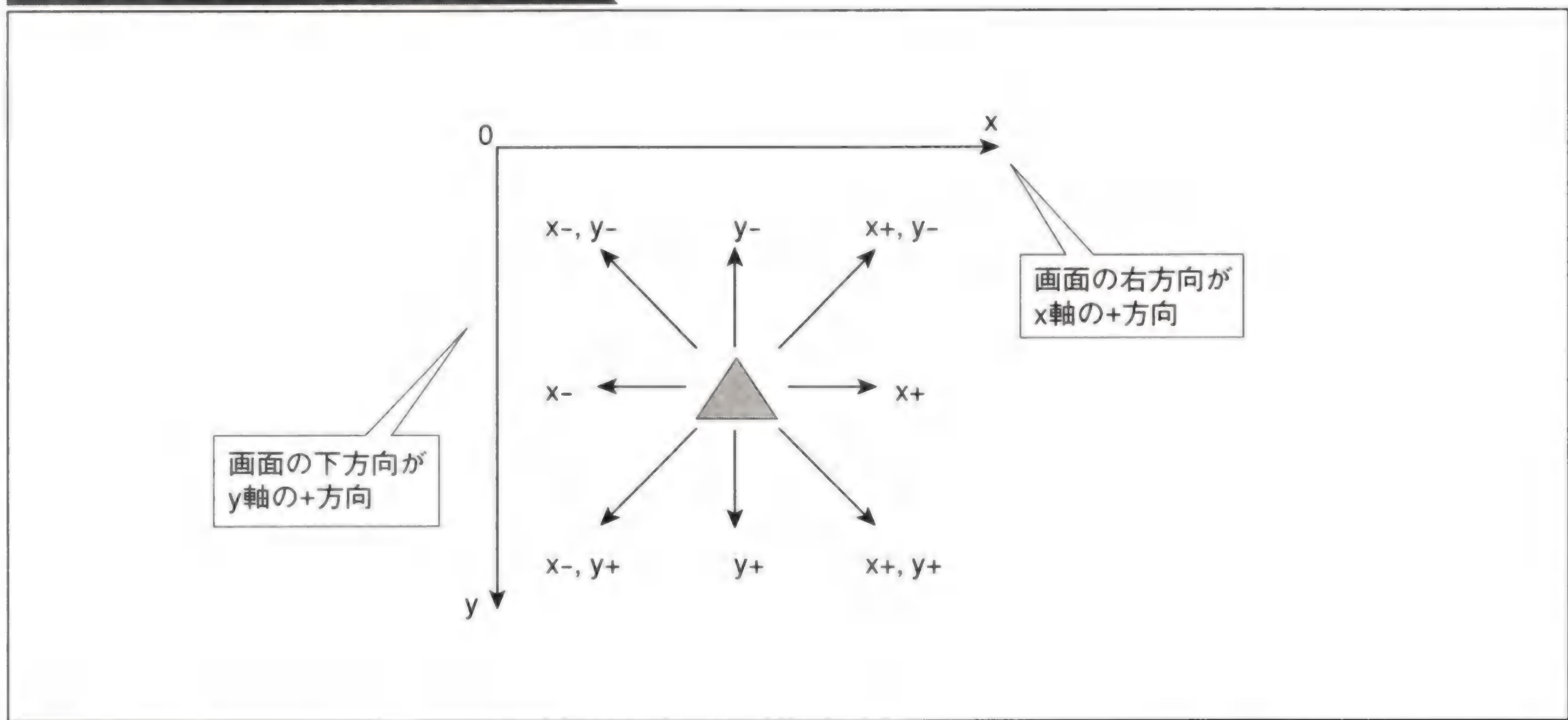
8方向に自機を移動させるには、スティックの入力に従って自機の座標を変化させます。スティックの入力を読み取る方式は、プログラミングに使う言語やライブラリによって異なりますが、たいていは上下左右のそれぞれが入力されたかどうかがYesまたはNo（trueまたはfalse）でわかるようになっています。

自機の座標を（x, y）とし、スティックの各方向が入力されたかどうかを変数up、down、left、

rightで表すと、自機を移動するプログラムはList 3-1のようになります。ここではFig. 3-2のように、画面の右方向がx軸の+方向、下方向がy軸の+方向になるように座標軸をとりました。これはコンピュータグラフィックにおける一般的な座標軸のとりかたです。

自機が移動する速さは変数speedで表しました。ここではx、y、speedを実数型（float）にしましたが、整数型（int）を使う方法もあります。

スティックを斜めに入力したときにはxとyの両方が変化し、結果として自機は斜めに動きます。座標軸に沿った4方向に斜め4方向を加えて、全部で8方向に移動できるというわけです。

Fig. 3-2 画面の座標軸と自機の移動

List 3-1 自機の移動

```
void MoveMyShip(
    float& x, float& y,     // 自機の座標（X方向, Y方向）
    float speed,            // 自機の速さ
    bool up, bool down,     // 上下方向へのスティック入力
    bool left, bool right   // 左右方向へのスティック入力
) {
    if (up   ) y-=speed;
    if (down ) y+=speed;
    if (left ) x-=speed;
    if (right) x+=speed;
}
```

自機の移動可能範囲

実際のゲームでは、自機がどこまでも動けるわけではなく、移動範囲が決められています。移動可能範囲は画面いっぱいのことも、もっと狭いこともありますが、いずれにしてもFig. 3-3のような矩形（長方形）枠で表すことができます。

自機の移動範囲をFig. 3-3のような枠内に制限する方法は、Fig. 3-4のとおりです。ここでは自機の左上座標を（x0, y0）、右下座標を（x1, y1）とします。枠の左上座標を（sx0, sy0）、右下座標を（sx1, sy1）とすると、自機が枠のなかに入るための条件は次のとおりです。

上：sy0<=y0
下：y1<=sy1
左：sx0<=x0
右：x1<=sx1

Fig. 3-4の条件をプログラムにしたものがList 3-2です。（x0, y0）や（x1, y1）などの値は自機の大きさに応じて座標（x, y）から算出します。たとえば、自機の座標が（100, 100）で、自機の幅が20、高さが10のときには次のようになります。

(x0, y0) = (x, y) = (100, 100)
(x1, y1) = (x+20, y+10) = (120, 110)

Fig. 3-3 自機の移動可能範囲

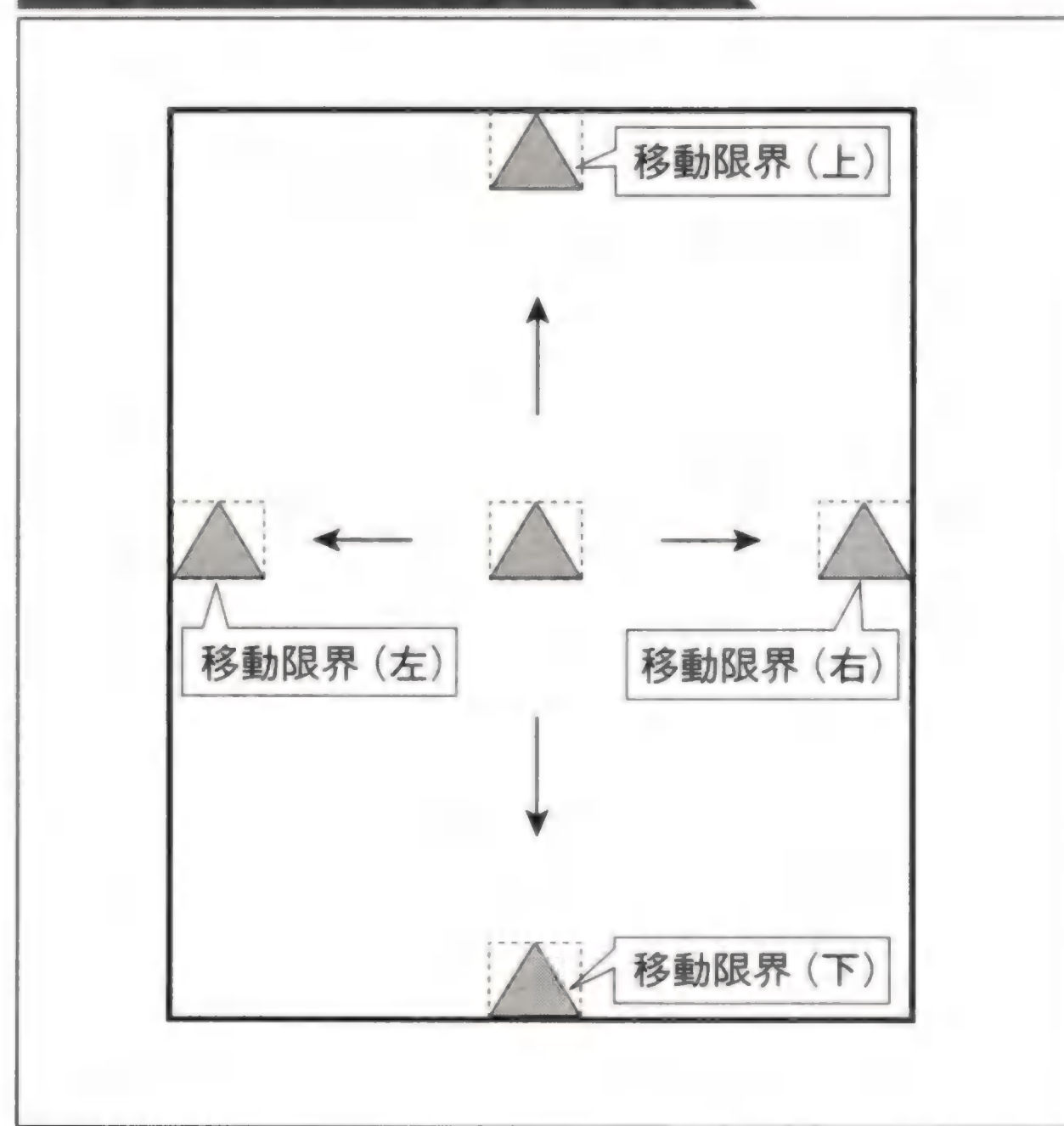
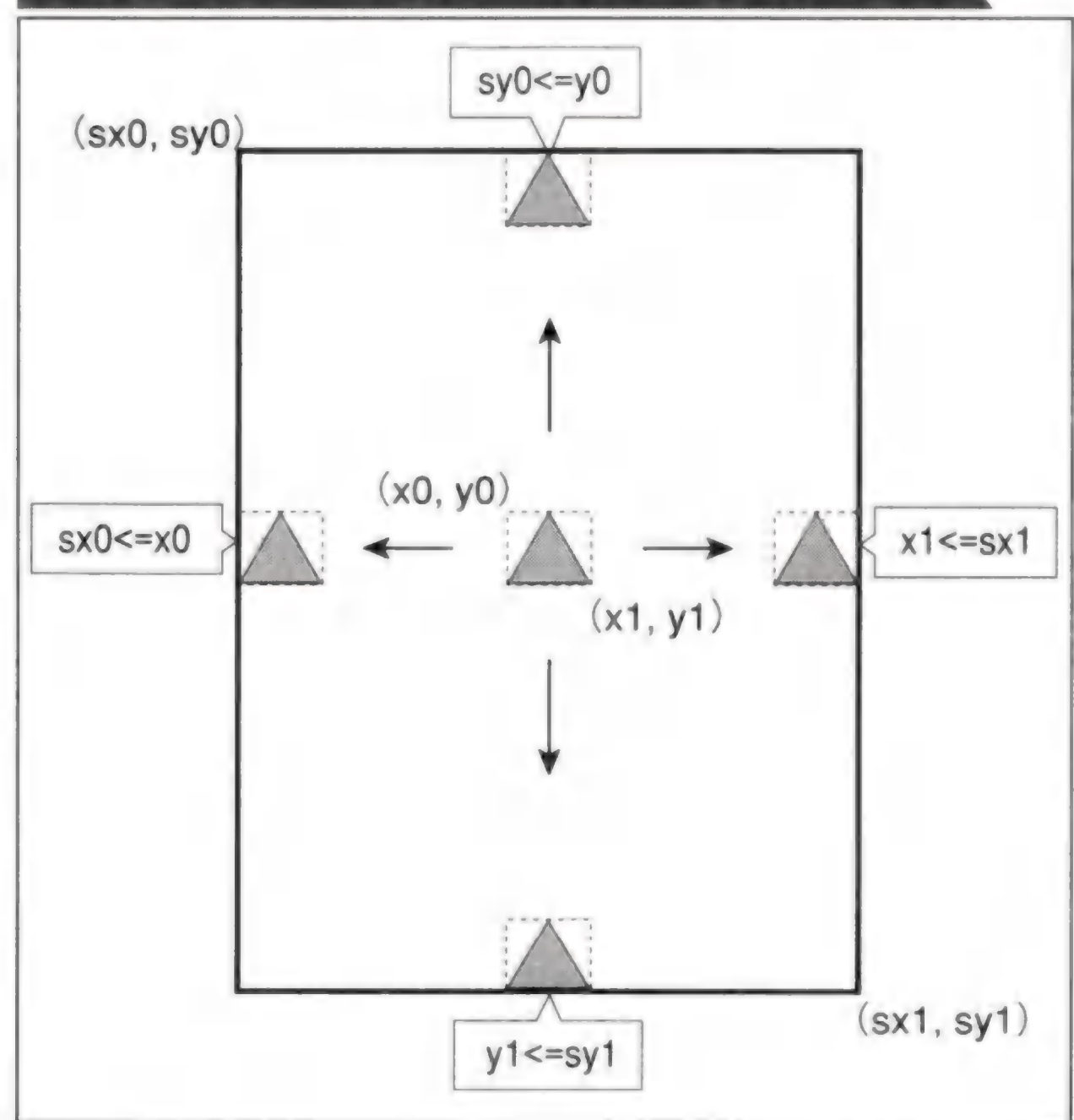



Fig. 3-4 自機が枠のなかに入るための条件

List 3-2 移動可能範囲を考慮した自機の移動

```
void MoveMyShip2(
    float& x, float& y,     // 自機の座標(X方向, Y方向)
    float speed,            // 自機の速さ
    float x0, float y0,     // 自機の左上座標
    float x1, float y1,     // 自機の右下座標
    float sx0, float sy0,   // 移動可能枠の左上座標
    float sx1, float sy1,   // 移動可能枠の右下座標
    bool up, bool down,     // 上下方向へのスティック入力
    bool left, bool right   // 左右方向へのスティック入力
) {
    if (up    && sy0<=y0) y-=speed;
    if (down  && y1<=sy1) y+=speed;
    if (left  && sx0<=x0) x-=speed;
    if (right && x1<=sx1) x+=speed;
}
```

ロールの表示

現実の飛行機が旋回する場合、旋回する方向に機体がロールします(横方向に傾く)。多くのシューティングゲームでも同じように、自機が動くときには「自機がロールしているかのような表示」を行います。

Fig. 3-5はトップビュー(上から見る視点。縦スクロールゲームなど)の場合のロール表示です。左に移動するときには自機を左に傾け、右に移動するときには右に傾けます。Fig. 3-6はサイドビュー(横から見る視点。横スクロールゲームなど)の場合のロール表示です。

Fig. 3-5 トップビューの場合のロール表示

通常の状態
左に移動するときには左にロールする
右に移動するときには右にロールする

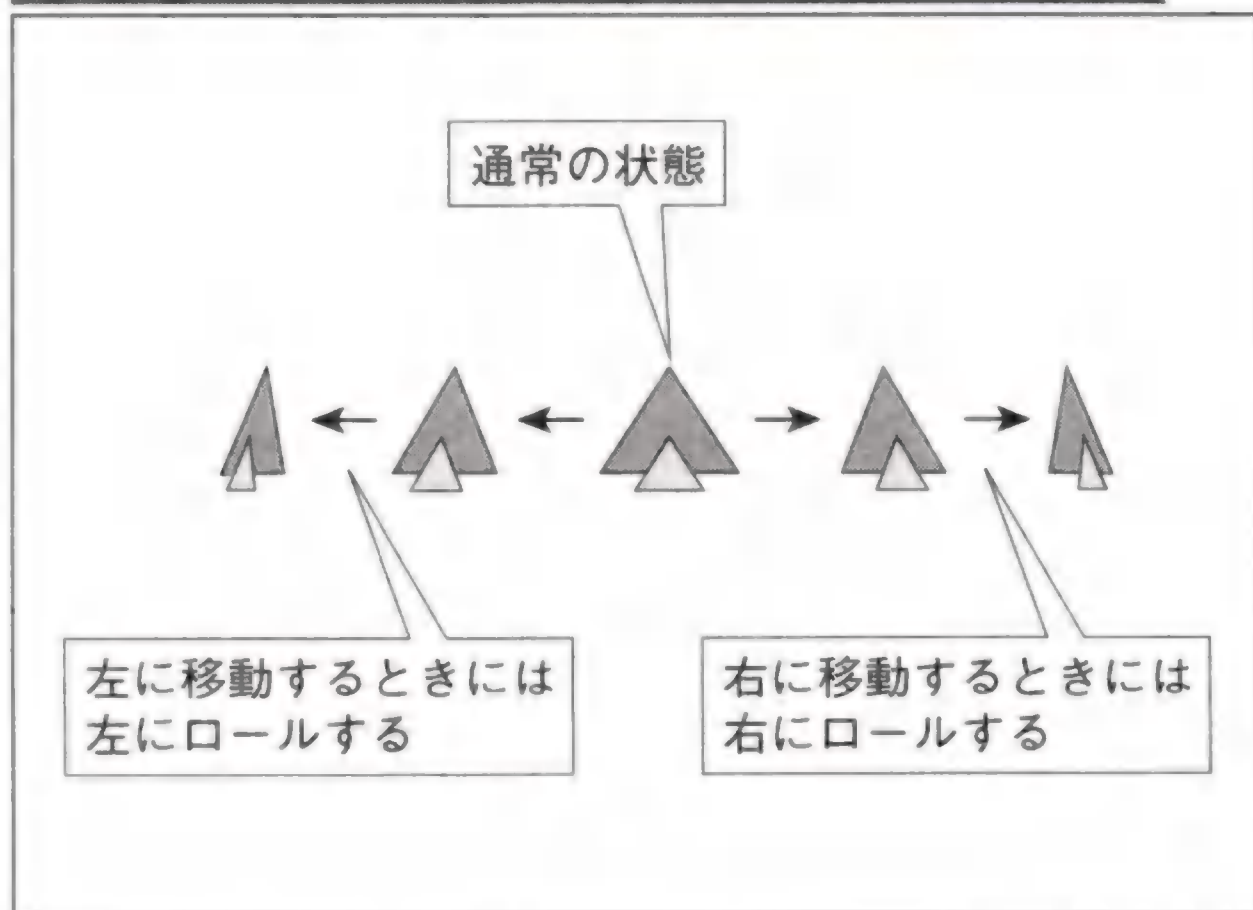
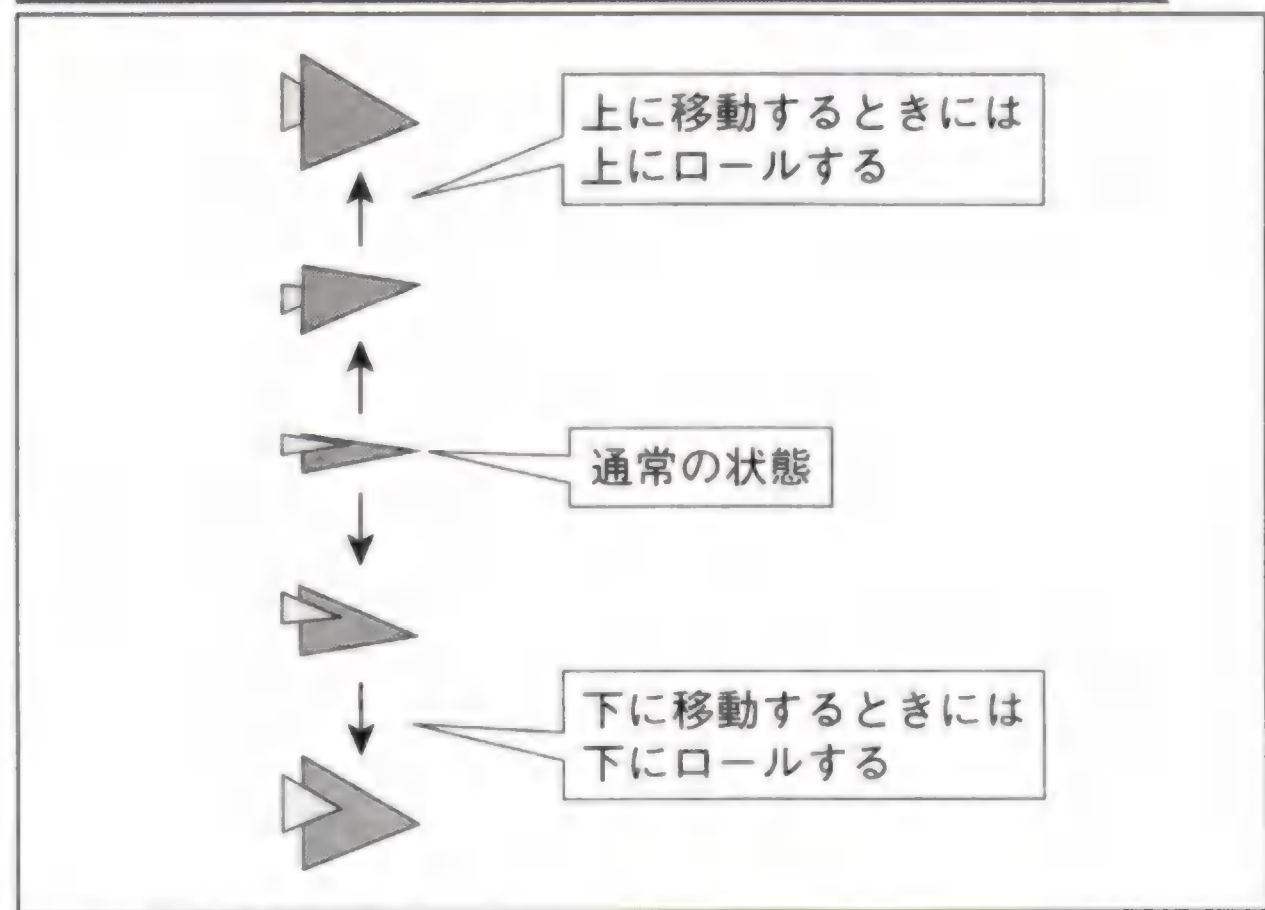



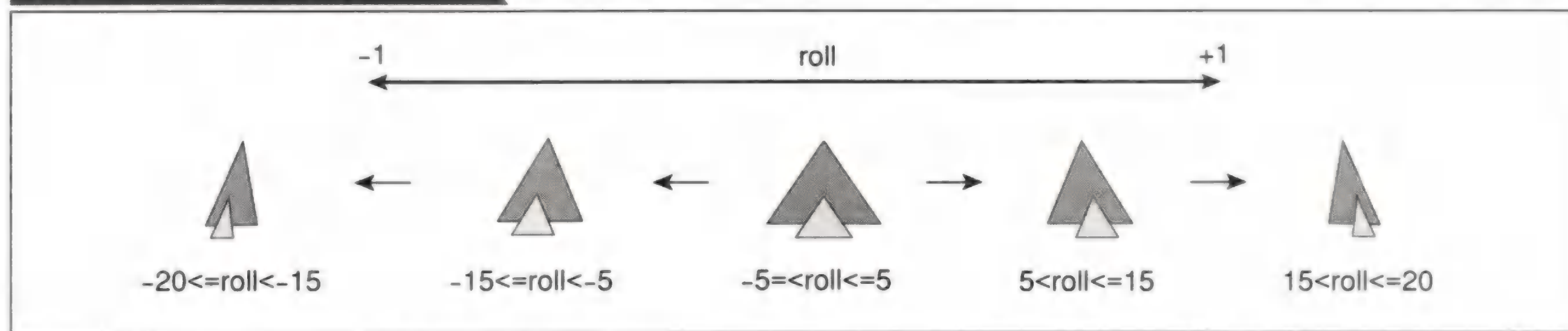
Fig. 3-6 サイドビューの場合のロール表示

上に移動するときには上にロールする
通常の状態
下に移動するときには下にロールする

上下に移動するときは、自機をそれぞれ上下に傾けます。

ロール表示をするにはFig. 3-7のような方法を使います。機体がどれだけロールしているかを保存する変数を用意して、その変数の範囲によって表示するパターンを変えます。ここでは変数をrollとしました。左に移動するときにはrollの値を−1して、右に移動するときには+1します。

List 3-7 ロール表示の方法

-1 roll +1

-20<=roll<-15　-15<=roll<-5　-5=<roll<=5　5<roll<=15　15<roll<=20

ここでは、自機がロールするパターンを左右2つずつ用意してあるとします。この場合、たとえば−15<=roll<−5のときには小さく左にロールしたパターンを表示し、−20<=roll<−15のときには大きく左にロールしたパターンを表示します。

スティックを左右に倒していないときには、rollの値を0に近づくように変化させます。roll<0のときには+1して、roll>0のときには−1するわけです。こうすれば、スティックを倒さないでいると自機が自然にロールしていない状態に戻っていきます。

変数rollの値の範囲とパターンとの対応づけによって、自機がロールする速さが変わります。値の範囲を狭くするとロールが速くなり、広くすると遅くなります。このあたりは実際にゲームを動かしながら、見た目がよくなるように値の範囲を調整する必要があります。

Fig. 3-7の方法でロール表示をするプログラムはList 3-3のようになります。ここでは自機の移動や移動範囲の制限に関する処理は除いて、ロール表示に関する処理だけを書きました。また、パターンの表示に関する具体的な処理は関数Drawのなかで行うものとしました。

List 3-3 ロール表示を行うプログラム

```
void RollMyShip(
    int& roll,              // ロールの角度
    bool left, bool right   // 左右方向へのスティック入力
) {
    // スティックで左を入力した場合：
    // -20<rollならばrollを-1する。
    if (left && -20<roll) roll--;

    // スティックで右を入力した場合：
    // roll<20ならばrollを+1する。
    if (right && roll<20) roll++;
```

```
    // スティックで左も右も入力していない場合：
    // rollが0に近づくようにする。
    if (!left && !right) {
        if (roll<0) roll++; else
        if (0<roll) roll--;
    }

   // rollの値に応じて異なるパターンを表示する：
    // パターン表示の具体的な処理は関数Drawのなかで行うとする。
    if (-20<=roll && roll<-15) Draw(LEFT_BIG_ROLL);     else
    if (-15<=roll && roll< -5) Draw(LEFT_SMALL_ROLL);   else
    if ( -5<=roll && roll<= 5) Draw(CENTER);            else
    if (  5< roll && roll<=15) Draw(RIGHT_SMALL_ROLL);  else
    if ( 15< roll && roll<=20) Draw(RIGHT_BIG_ROLL);
}
```

■ 3Dでのロール表示

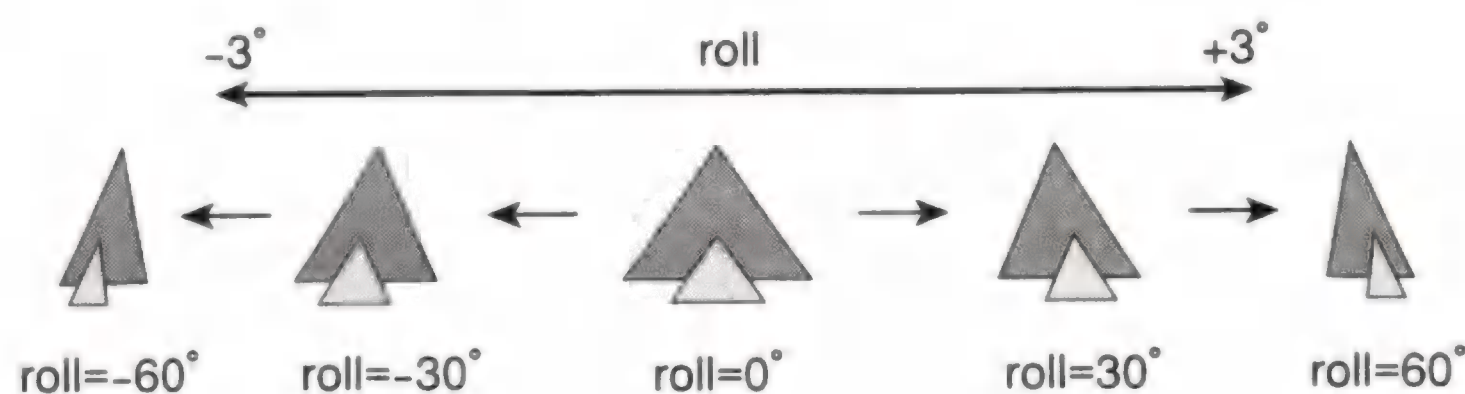
自機の表示に3Dグラフィックを使う場合には、3Dオブジェクトを回転表示することによってなめらかなロール表示ができます。プログラミングも簡単で、Fig. 3-8のようにロールの角度を保存しておき、その角度で自機を回転表示するだけですみます。Fig. 3-8では−60°、−30°、0°、30°、60°の場合のパターンを示しましたが、実際にはこれらの間の角度についても回転表示されるので、非常になめらかにロールします。

Fig. 3-8ではロールの角度をrollとしました。Fig. 3-7の場合と同じように、自機が左右に動いたときにはrollの値を変化させます。たとえば左に動いたときに−3°、右に動いたときに+3°といった要領です。値の変化が大きければロールが速くなり、小さければロールは遅くなります。

List 3-4は、Fig. 3-8の方法でロール表示を行うプログラムです。前半はList 3-3とあまり変わりませんが、後半の表示部分はずっと簡単になります。

Fig. 3-8 3Dグラフィックを使ったロール表示の方法

-3° roll +3°

roll=-60° roll=-30° roll=0° roll=30° roll=60°

List 3-4 3Dグラフィックを使ってロール表示を行うプログラム

```
void RollMyShip3D(
    int& roll,               // ロールの角度
    bool left, bool right   // 左右方向へのスティック入力
) {
    // スティックで左を入力した場合：
    // -60<rollならばrollを-3する。
    if (left && -60<roll) roll-=3;

    // スティックで右を入力した場合：
    // roll<60ならばrollを+3する。
    if (right && roll<60) roll+=3;

    // スティックで左も右も入力していない場合：
    // rollが0に近づくようにする。
    if (!left && !right) {
        if (roll<0) roll+=3; else
        if (0<roll) roll-=3;
    }

    // rollの値に応じて自機を回転させて表示する：
    // 表示の具体的な処理はDraw関数で行うとする。
    Draw(roll);
}
```

斜め移動の速さ

斜め移動は上下や左右の移動に比べて速くなることがあります。Fig. 3-9は上下方向、左右方向、斜め方向のそれぞれについて自機が移動する速さを示した図です。上下と左右に速さ1で移動するとした場合、斜め移動の場合には上下移動と左右移動の組み合わせになるので、移動の速さは約1.4倍（$1/\cos45^\circ$倍$=\sqrt{2}$倍）になります。

実数型を使う場合には、上下左右と斜めとで移動の速さを厳密に同じにすることもできます（Fig. 3-10）。このときは上下と左右に速さ1で移動し、斜めに移動する場合には上下方向と左右方向にそれぞれ約0.7（$\cos45^\circ=1/\sqrt{2}$）ずつ移動します。

実際のシューティングゲームでは、Fig. 3-9のように斜め移動が速くなる方式を採用していることが多いようです。そのためか、Fig. 3-10のように上下左右と斜めの速さを同じにすると、逆に斜め移動が遅くなったように感じられます。

List 3-5は上下左右と斜めの速さを同じにした場合のプログラムです。

Fig. 3-9 移動方向による速さの違い

$$\frac{1}{\cos 45^\circ} = \frac{1}{1/\sqrt{2}} = \sqrt{2} \cong 1.4$$

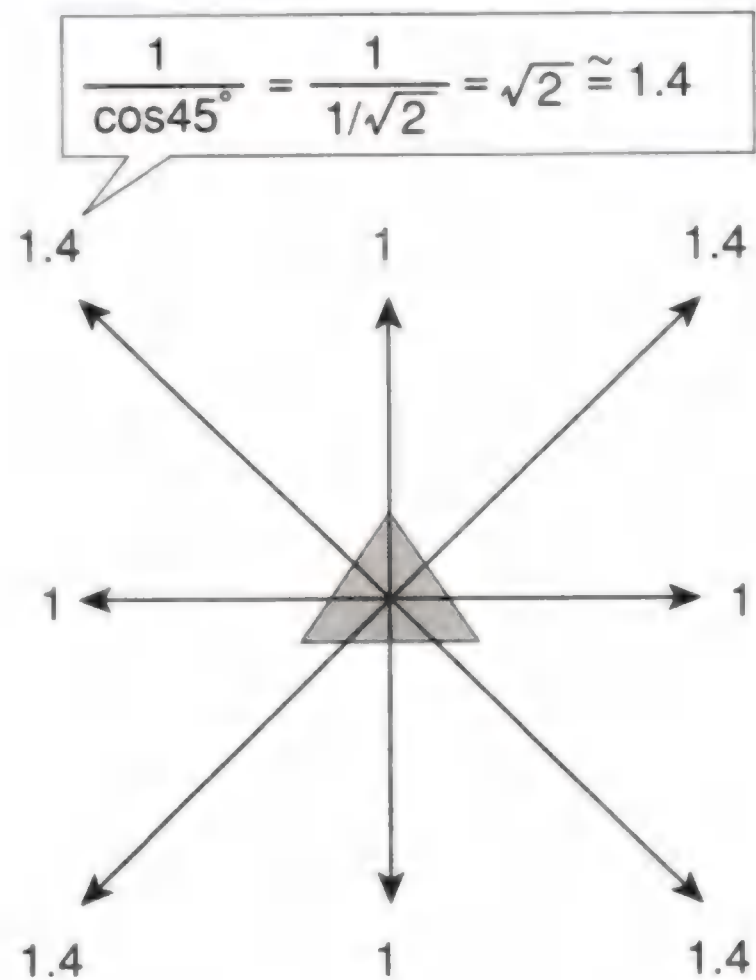
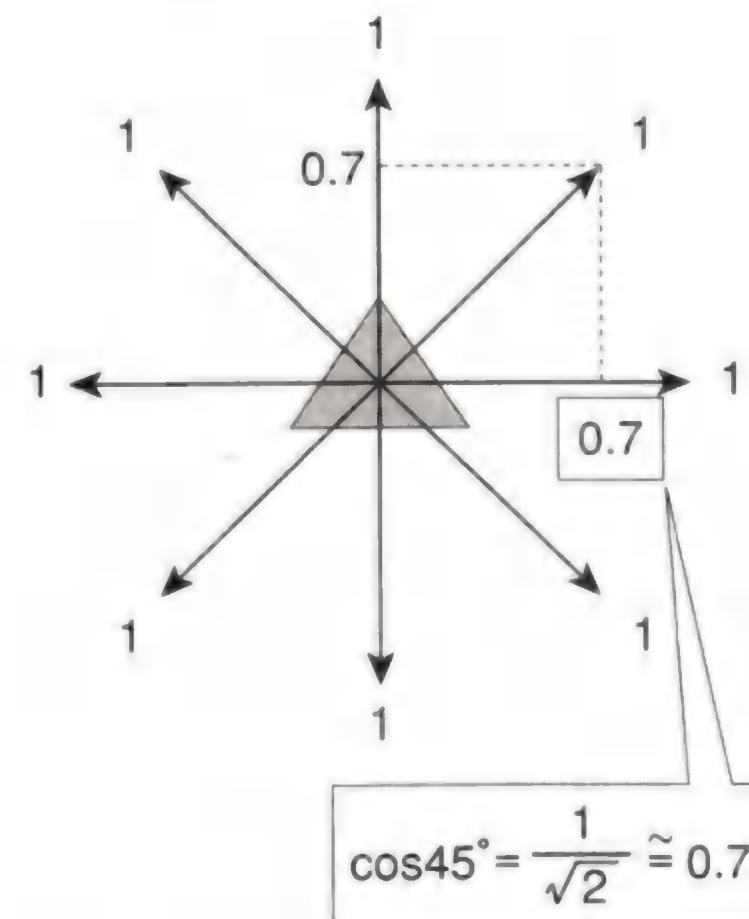



Fig. 3-10 上下左右と斜めの速さを同じにする

$$\cos 45^\circ = \frac{1}{\sqrt{2}} \cong 0.7$$

List 3-5 上下左右と斜めの速さを同じにした移動

```
#include <math.h>

void MoveAtConstantSpeed(
    float& x, float& y,     // 自機の座標(X方向, Y方向)
    float speed,            // 自機の速さ
    bool up, bool down,     // 上下方向へのスティック入力
    bool left, bool right   // 左右方向へのスティック入力
) {
    // 斜めに移動するときのスピード:
    // 上下左右の約0.7倍にする。
    float s=speed/sqrt(2);

    // 斜め移動
    if (up   && left ) x-=s, y-=s; else
    if (up   && right) x+=s, y-=s; else
    if (down && left ) x-=s, y+=s; else
    if (down && right) x+=s, y+=s; else

    // 上下左右移動
    if (up    && !down ) y-=speed; else
    if (down  && !up   ) y+=speed; else
    if (left  && !right) x-=speed; else
    if (right && !left ) x+=speed;
}
```

アナログスティックを使った移動

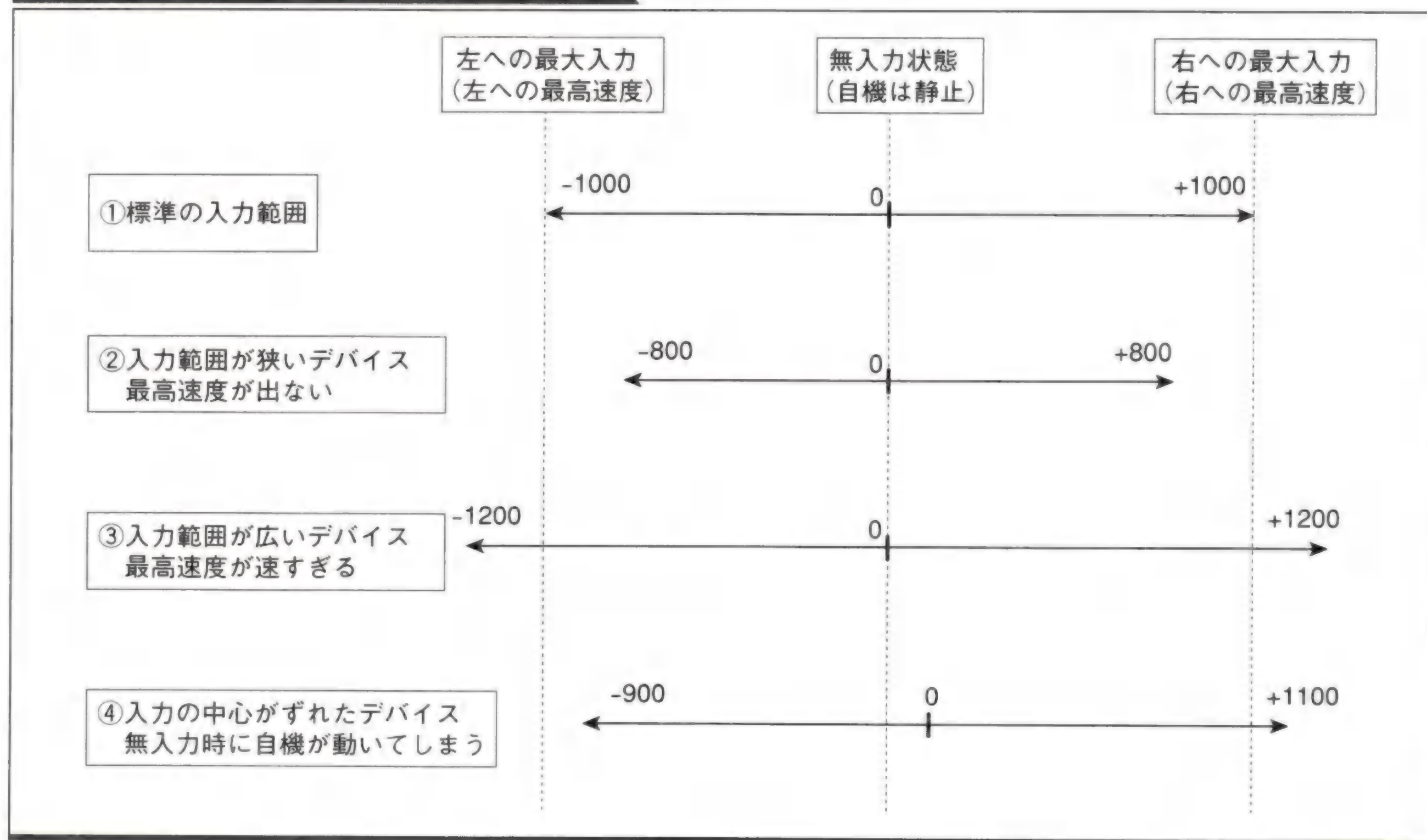
2Dシューティングゲームではあまり見かけませんが、アナログスティックを使って自機を移動させることもできます。アナログスティックを使ったアナログ入力には、デジタル入力に比べて次のようなメリットがあります。

- **移動速度を細かく調節できる**
- **移動方向を8方向以上に細かく調節できる**

速度や方向が細かく調整できるので、デジタル入力のゲームよりも細やかな弾よけができるかもしれません。デジタル入力は自機を細かく動かすときには小刻みにスティックを倒したり起こしたりしますが、アナログ入力ではスティックの傾きで移動量をコントロールすることができます。アナログ入力のシューティングゲームはまだ一般的ではありませんが、工夫しだいで、まったく新しいテイストのゲームができるかもしれません。

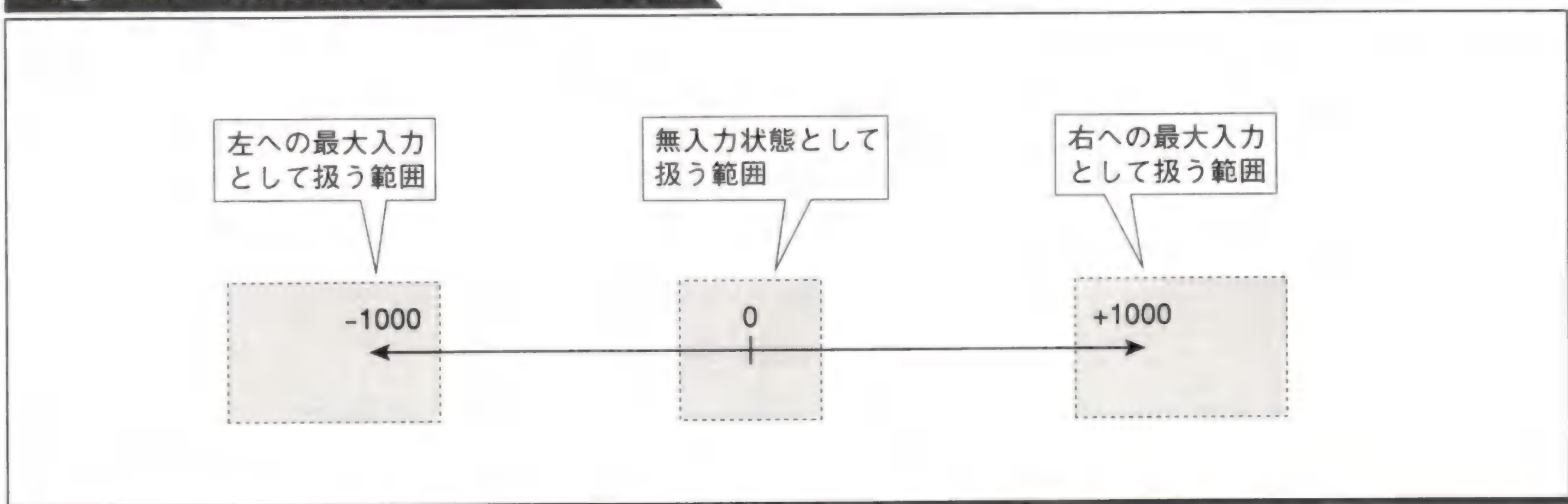
アナログ入力値と「あそび」

アナログ入力を使うときには、デバイス（スティックなどの装置）の種類によって入力値の範囲が違うことがあるのに気をつけなければなりません。うっかり上限いっぱいの値まで使ってしまうと、デバイスによって自機の最高速度が変わってしまうことがあります（Fig. 3-11）。

Fig. 3-11 デバイスによる入力範囲の違い

アナログ入力では、デバイスから入力された値を自機の速度に変換します。しかし入力値の範囲はデバイスによって違うので、たとえば入力範囲が狭いデバイス（Fig. 3-11-②）では、最高速度が遅くなってしまいます。逆に入力範囲が広いデバイス（Fig. 3-11-③）では、想定された最高速度よりも速い速度が出てしまうかもしれません。

また、無入力状態（スティックをまったく倒さない状態）の扱いにも注意が必要です。無入力状態での入力値はスティックによってばらつきがあります。そのため、入力値0だけを無入力状態として扱うようなプログラムでは、中心がずれたスティック（Fig. 3-11-④）を使ったとき、何も入力していないのに自機が動いてしまいます。

こういった入力範囲の違いを吸収するには、入力の最大値や入力の中心の周りにマージン（余裕）を設けて、最大値や中心として扱う範囲をやや広めにとります（Fig. 3-12）。

Fig. 3-12 最大値や中心の範囲を広めにとる

なお、入力の中心を広めにとることを「あそび」とも呼びます。車を運転する方はご存じかと思いますが、現実の車のハンドルをほんの少し切っただけでは車は曲がらないでしょう。これは、ハンドルにも「あそび」があるからです。ゲームでも同じことで、スティックをほんの少し倒しただけでは自機が動かないようにしておいたほうが、操作がしやすくなります。

■ 入力値から移動量への変換

アナログスティックを使って自機を移動させるには、スティックの入力値を自機の移動量に変換する必要があります。たとえばX方向（横方向）の移動について、スティックの入力値をjx、入力値の範囲を−rangeから+rangeまで、自機の最高速度をspeedとすると、自機の座標xを更新する式は次のようになります。

```
x+=speed*jx/range
```

これは入力範囲に余裕（あそび）を持たせていないので、Fig. 3-13のように入力範囲を広くします。この場合には、入力値jxによって次のように座標xを更新します。

- **jx<=-range+marginのとき（左への最大入力）：x-=speed**
- **-margin<=jx && jx<=marginのとき（無入力）：x+=0**

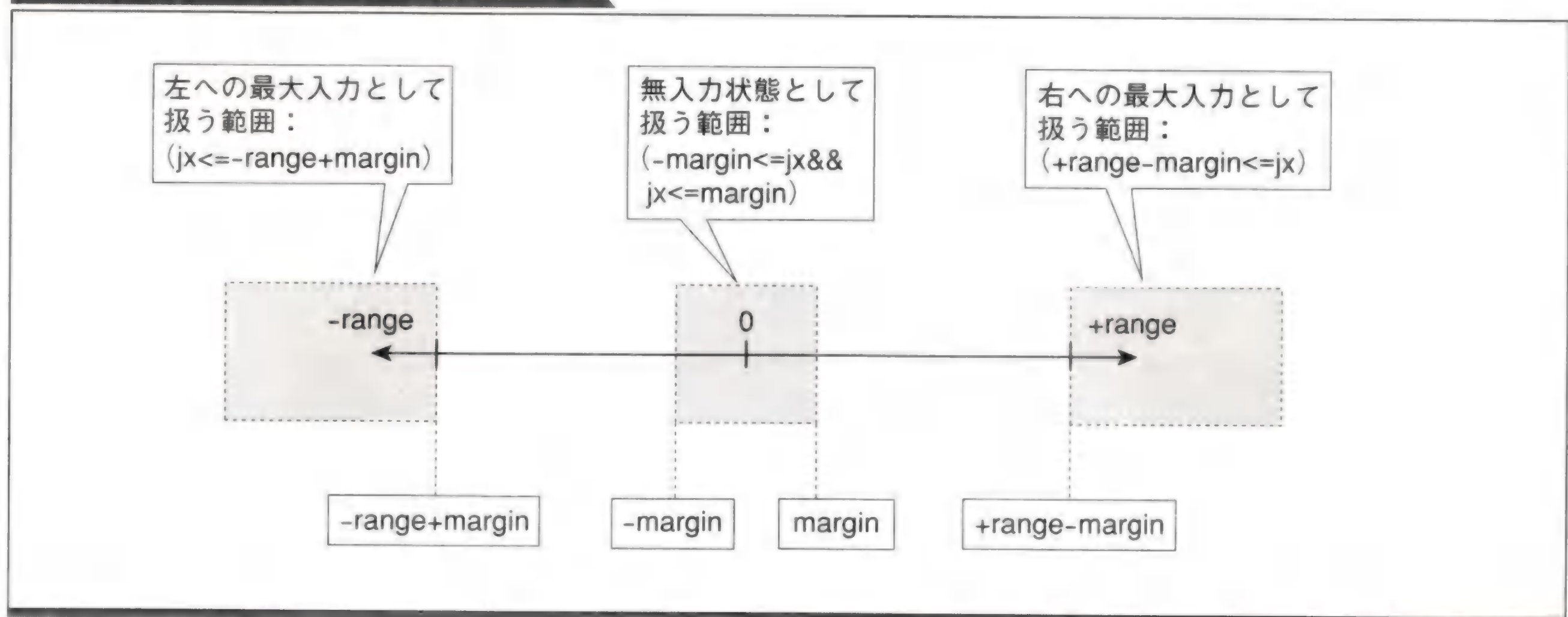
・+range-margin<=jxのとき（右への最大入力）：x+=speed

上記の3条件以外のときには、入力値の大きさ（スティックの傾きの度合い）によって自機をアナログ的に移動させます。

・jx<0のとき（左への入力）：x+=speed*（jx+margin）/（range-margin*2）
・jx>0のとき（右への入力）：x+=speed*（jx-margin）/（range-margin*2）

上記の2つの式では、最大値と中心のマージンを除いた範囲について、入力値を-1～1までの値に変換（比例配分）し、それにspeedを掛けた値をxに加えています。

Fig. 3-13 余裕を持たせた入力範囲

左への最大入力として扱う範囲：（jx<=-range+margin）
無入力状態として扱う範囲：（-margin<=jx&&jx<=margin）
右への最大入力として扱う範囲：（+range-margin<=jx）
-range
0
+range
-range+margin
-margin
margin
+range-margin

以上の式をプログラムにまとめたものがList 3-6です。なお、付録CD-ROMのすべてのサンプルでは、アナログスティックを使って自機を移動することができます。実際にプレイしてみると、細かい弾よけはデジタルよりもアナログのほうがやりやすいことがあります。「アナログスティックを使って極端に細かい弾よけをさせるゲーム」などを作ってみても面白いかもしれません。

List 3-6 アナログ入力で自機を移動させるプログラム

```
void MoveByAnalog(
    float& x, float& y,       // 自機の座標（X方向、Y方向）
    float speed,              // 自機の最高速度
    int jx, int jy,           // スティックの入力値（X方向、Y方向）
    int range, int margin     // 入力値の範囲（最大値）、マージン
) {
    // X方向に関する移動
    if (jx<=-range+margin) x-=speed; else
    if (+range-margin<=jx) x+=speed; else
    if (jx<-margin) x+=speed*(jx+margin)/(range-margin*2); else
```

```
    if (+margin<jx) x+=speed*(jx-margin)/(range-margin*2);

    // Y方向に関する移動
    if (jy<=-range+margin) y-=speed; else
    if (+range-margin<=jy) y+=speed; else
    if (jy<-margin) y+=speed*(jy+margin)/(range-margin*2); else
    if (+margin<jy) y+=speed*(jy-margin)/(range-margin*2);
}
```

ワープ移動

特殊なスティック入力によって、自機が一瞬にして離れた場所へ移動するものです。この移動方法は「式神の城（→ P. 327）」の一部のキャラクターなどに見られます。「式神の城」の場合はスティックを素早く同じ方向へ2回入力することによって、自機がワープします（Fig. 3-14）。ワープ移動は特殊な移動方法ですが、上手に使えば普通では回避不可能な弾幕を抜けることも可能です（Fig. 3-15）。

「スティックが同じ方向へ2回入力された」というのは、次のようなことを意味します（Fig. 3-16）。

Fig. 3-14 ワープ移動

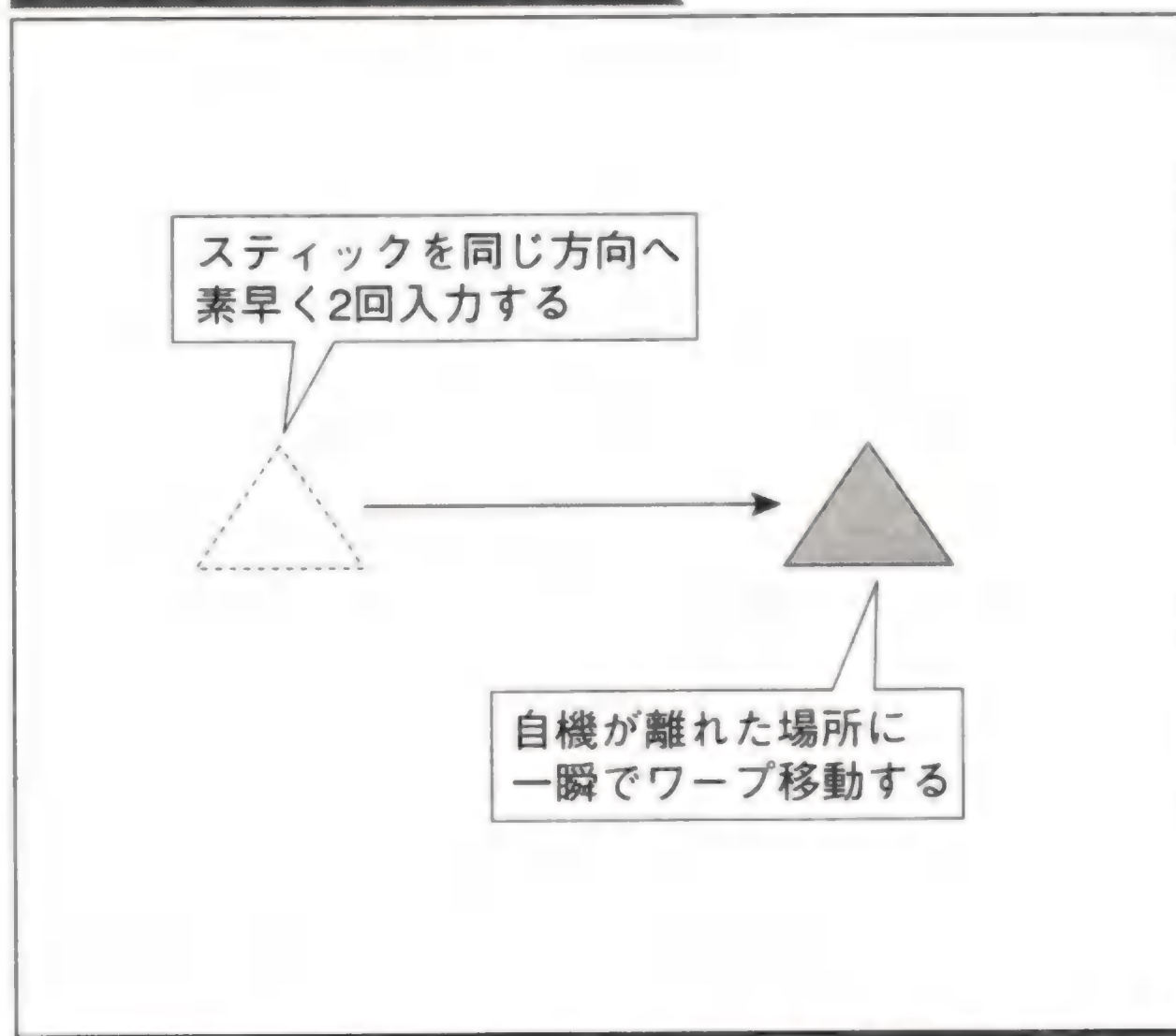
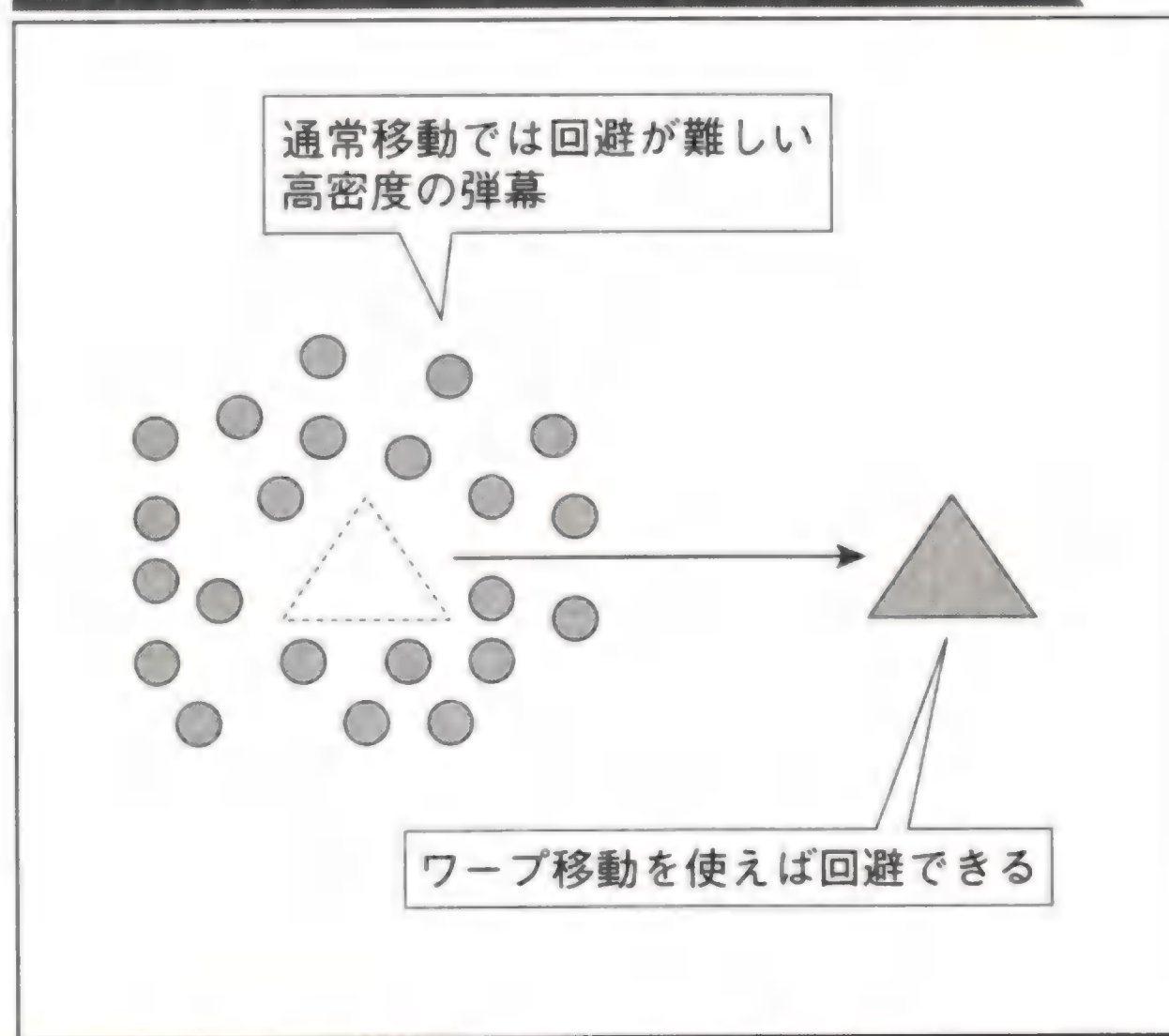
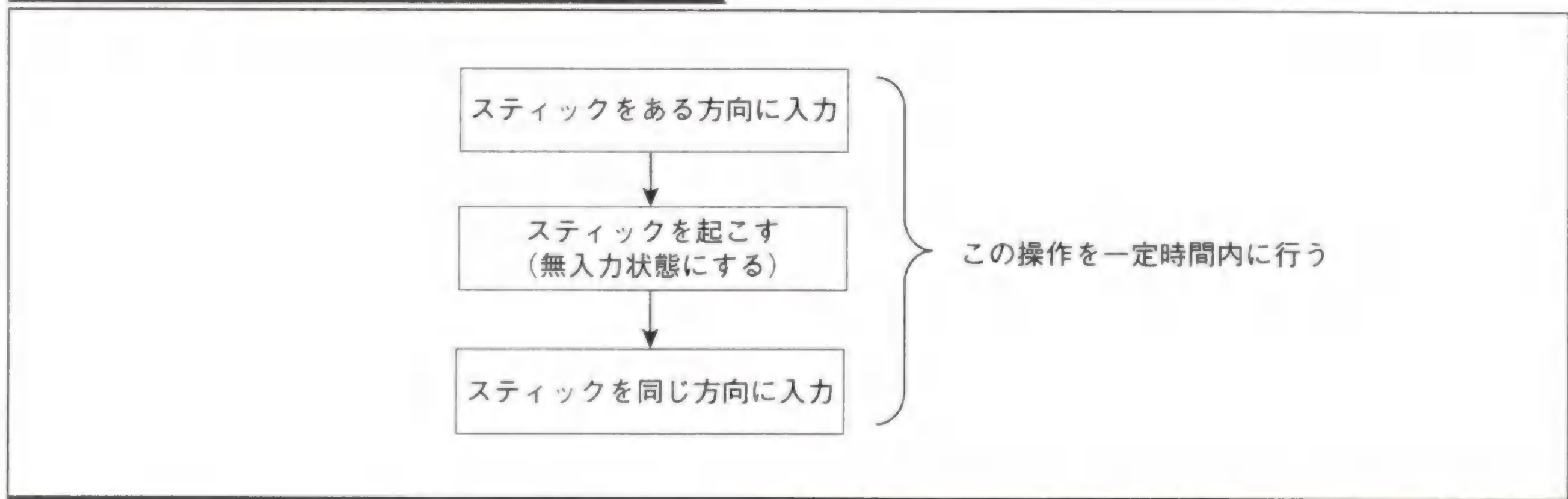



Fig. 3-15 ワープ移動を利用した弾幕回避

①スティックをある方向に入力してから
②いったんスティックを中央に戻し
③再び同じ方向に入力した

この操作を一定時間内に行ったときに、スティックが同じ方向に「素早く」2回入力されたと判定します。

ワープ移動のプログラムはList 3-7のようになります。もしもワープ移動をもっと簡単に入力したい場合、たとえば同じ方向を2回入力する間に別の方向の入力が入ってもかまわなくするような場合には、「コマンドショット」(→ P. 131) と同じ方法でスティック入力を判定します。

Fig. 3-16 スティックを同じ方向に入力する

サンプル

ワープ移動 → P. 315

List 3-7 ワープ移動

```
void Warp(
    float& x, float& y,     // 自機の座標
    float speed,            // 自機の移動速度
    float warp_dist,        // ワープの移動距離
    bool up, bool down,     // スティック入力(上下)
    bool left, bool right   // スティック入力(左右)
) {
    static bool
        up0=false, down0=false,       // 前回のスティック入力
        left0=false, right0=false,    // (上下、左右)
        released=true;                // 無入力状態にしたかどうか
    static int time=0;                // 入力の制限時間
```

```
    // スティックが入力されたとき
    if (up || down || left || right) {

        // ワープする場合：
        // 無入力をはさんで、かつ制限時間内に、
        // 前回と同じ方向が入力されたとき。
        if (released && time>0 &&
            up==up0 && down==down0 &&
            left==left0 && right==right0
        ) {
            if (up    ) y-=warp_dist;
            if (down  ) y+=warp_dist;
            if (left  ) x-=warp_dist;
            if (right) x+=warp_dist;
        }

        // ワープしない場合：
        // 無入力をはさんでいないか、制限時間切れか、
        // 前回と異なる方向が入力されたとき。
        // 方向を記録し、制限時間を設定する。
        else {
            up0=up; down0=down;
            left0=left; right0=right;
            time=10;

            // 普通に移動する
            if (up    ) y-=speed;
            if (down  ) y+=speed;
            if (left  ) x-=speed;
            if (right) x+=speed;
        }

        released=false;
    }

    // スティックが無入力状態のとき
    else released=true;

    // 時間を減らす
    if (time>0) time--;
}
```

ボタンによる自機のスピード調節

ボタンを使って自機の移動スピードを調節するというものです。自機のスピード調節があるゲームには、このようにボタンを使って自由に速度を調節できるものと、アイテムを取る（あるいはパワーアップする）ことによって速度を調節できるものとがあります（Fig. 3-17）。

ボタンを使ってスピードを調節するゲームでは、ボタンが1つの場合と2つの場合とがあります。ただ、多くのシューティングゲームではボタンの数を2つか3つにしぼることが多いので、スピード調節に2つもボタンを割り当てているゲームはまれです。

1つのボタンでスピードを調節する多くのゲームでは、ボタンを押すたびにスピードが上がり、最高速のときにボタンを押すと最低速に戻ります。List 3-8はこのようなスピード調節を行うプログラムです。最高速のときの処理と、ボタンを離したことを検出する処理がポイントになります。

Fig. 3-17 自機のスピード調節

	ボタンによるスピード調節（ボタンを1つ使用）	ボタンによるスピード調節（ボタンを2つ使用）	アイテムによるスピード調節（パワーアップによるスピード調節）
スピードアップの方法	ボタンを押す	スピードアップボタンを押す	スピードアップアイテムを取る（またはパワーアップでスピードアップを選択する）
スピードダウンの方法	最高速のときにボタンを押す（最低速になる）	スピードダウンボタンを押す	スピードダウンアイテムを取る（スピードダウンできないゲームもある）

サンプル

ボタンによるスピード調整 → P. 315

List 3-8 ボタンによるスピード調節

```
void SpeedControlByButton(
    float& speed,      // スピード
    float accel,       // スピードアップの度合い
    float max_speed,   // 最高速
    float min_speed,   // 最低速
    bool button        // 速度調節ボタンの入力
) {
```

```
    // 前回のボタンの状態
    static bool prev_button=false;

    // ボタンを一度離して押した場合：
    // 速度調節を行う。
    if (!prev_button && button) {
        if (speed>=max_speed) speed=min_speed;
        else speed+=accel;
    }
    // ボタンの状態を記録する
    prev_button=button;
}
```

アイテムによる自機のスピード調節

アイテムを取ってスピードを調節するというものです（Fig. 3-18）。スピードアップしかできないゲームと、スピードアップとスピードダウンの両方ができるゲームとがあります。また、スピードアップアイテムやスピードダウンアイテムを取るだけで自機の速度が変化するゲームと、パワーアップアイテムを取ったうえでスピードアップを選択することによって速度が変化するゲームとがあります。アイテムを拾ったかどうかを判定するには、自機に「拾い判定」を設定しておいて、アイテムとの当たり判定処理を行います（Fig. 3-19）。

Fig. 3-18 アイテムによるスピード調節

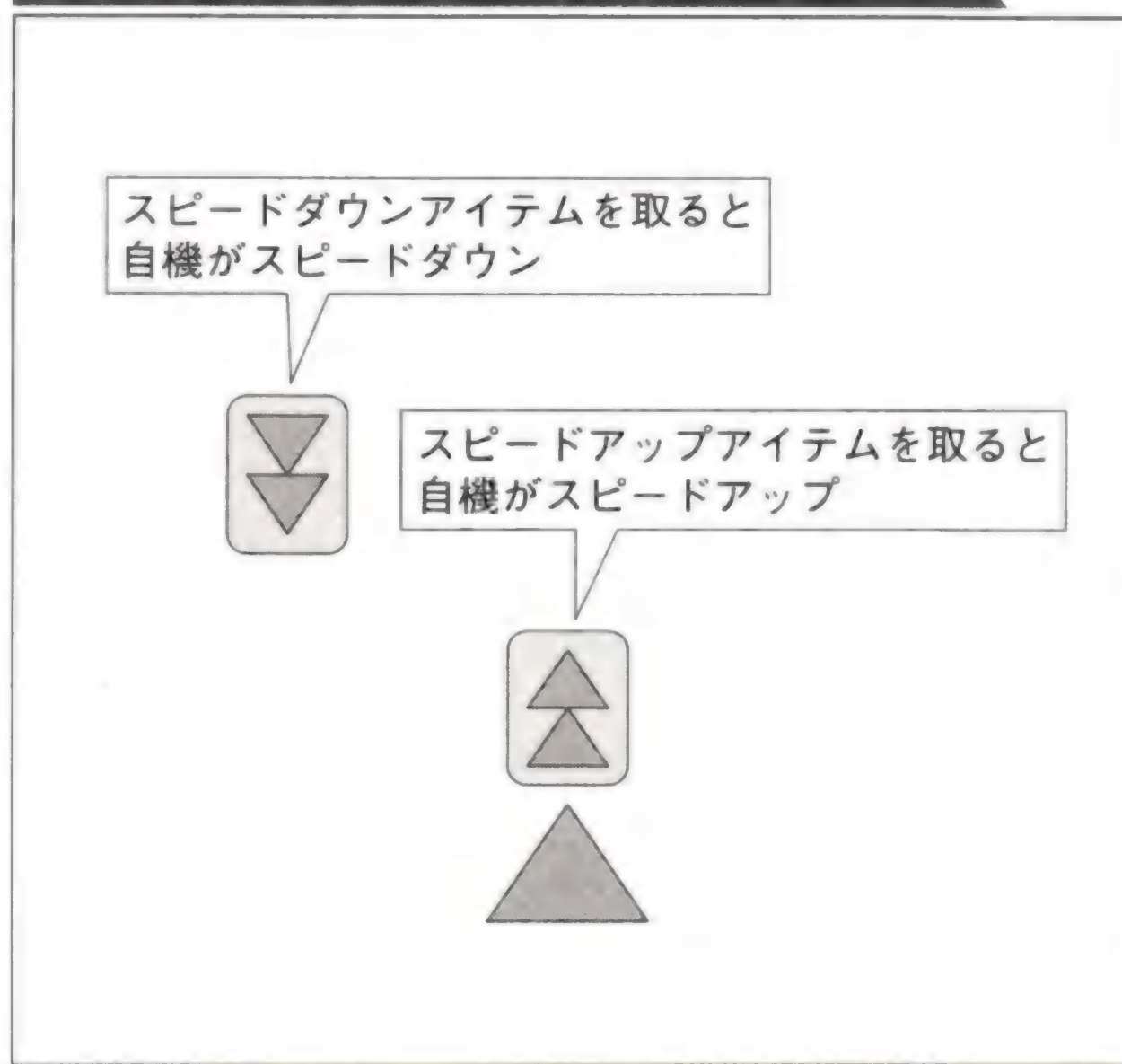
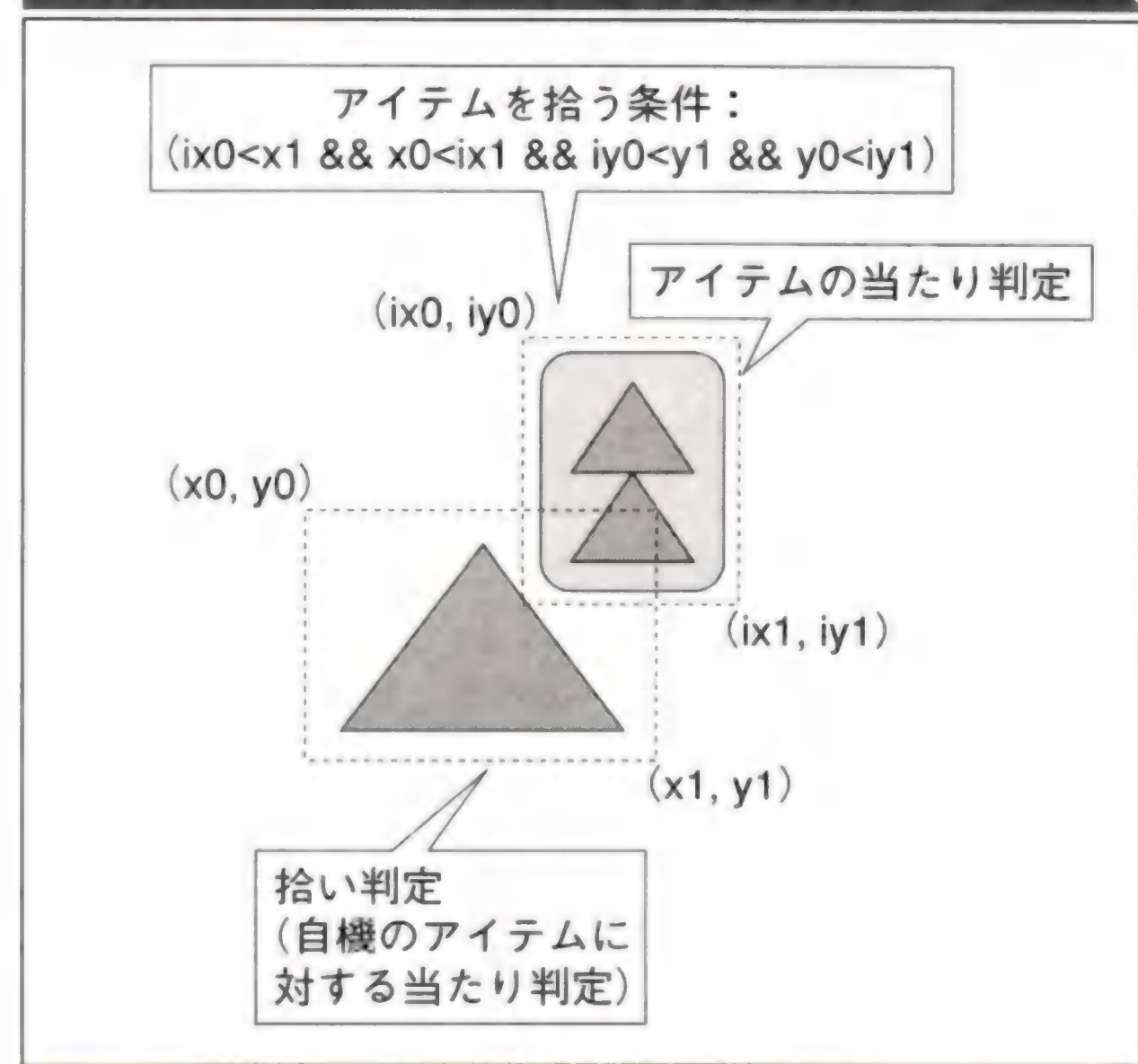



Fig. 3-19 アイテムと自機との当たり判定処理

拾い判定は「アイテムに対する自機の当たり判定」です（当たり判定 → P. 31）。アイテムはある程度拾いやすくする必要があるので、拾い判定は「弾や敵などに対する自機の当たり判定」よりも大きめにしておきます。あるいはアイテム側の当たり判定を大きくします。

拾い判定の左上座標を（x0, y0）、右下座標を（x1, y1）、アイテムの当たり判定の左上座標を（ix0, iy0）、右下座標を（ix1, iy1）とすると、アイテムを拾うための条件は次のように求まります。

(ix0<x1 && x0<ix1 && iy0<y1 && y0<iy1)

List 3-9はアイテムによるスピード調節のプログラムです。少しアレンジすると、スピードダウンアイテムを1個取っただけでスピードが最低になるようにもできます。

サンプル

アイテムによるスピード調整 → P. 315

List 3-9 アイテムによるスピード調節

```
void SpeedControlByItem(
    float& speed,            // スピード
    float max_speed,         // 最高速
    float min_speed,         // 最低速
    float x0, float y0,      // 拾い判定の左上座標
    float x1, float y1,      // 拾い判定の右下座標
    float ix0, float iy0,    // アイテムの当たり判定の左上座標
    float ix1, float iy1,    // アイテムの当たり判定の右下座標
    float accel              // スピードアップ・ダウンの度合い
) {
    // アイテムを拾ったときの処理：
    // 当たり判定処理を行い、当たったときには、
    // スピードアップまたはスピードダウンを行う。
    if (ix0<x1 && x0<ix1 && iy0<y1 && y0<iy1) {
        speed+=accel;

        // 以下のようにすると、
        // スピードダウンを1個拾っただけで、
        // スピードが最低になるようにできる
        // if (accel<0) speed=min_speed;

        // スピードが最高速や最低速を超えたら補正する
        if (speed>max_speed) speed=max_speed;
        if (speed<min_speed) speed=min_speed;
    }
}
```

合体する自機

自機が味方と合体してパワーアップするというものです（Fig. 3-20、3-21）。こうした合体は「ムーンクレスタ（→ P. 334）」「テラクレスタ（→ P. 331）」「Bウィング（→ P. 332）」などに見ることができます。合体することによって自機のショットがパワーアップしますが、かわりに自機の当たり判定も大きくなって、被弾しやすくなります。

Fig. 3-20 合体する自機

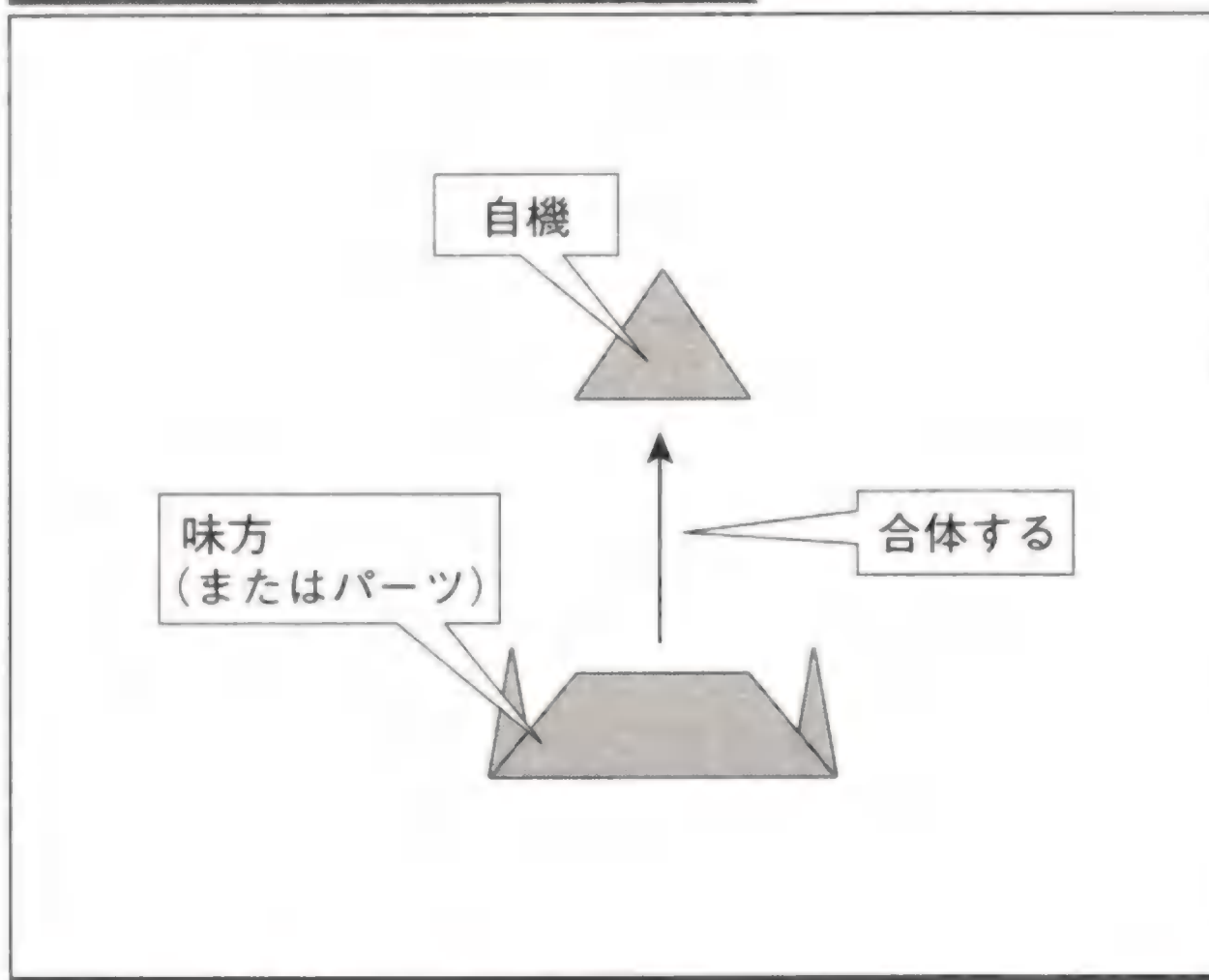
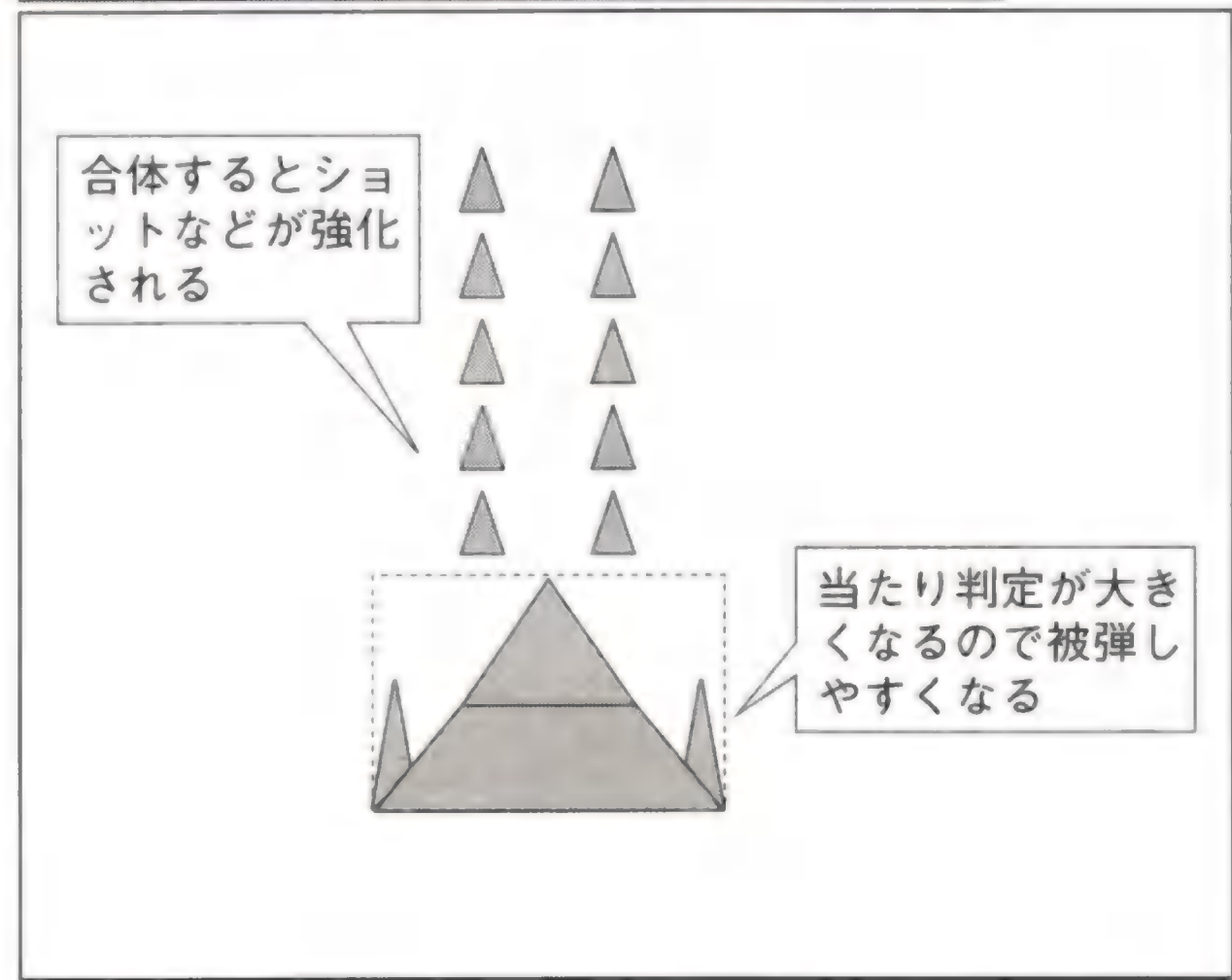



Fig. 3-21 合体によるパワーアップ

合体の方法はゲームによって違いますが、たとえば次のような手順があります。

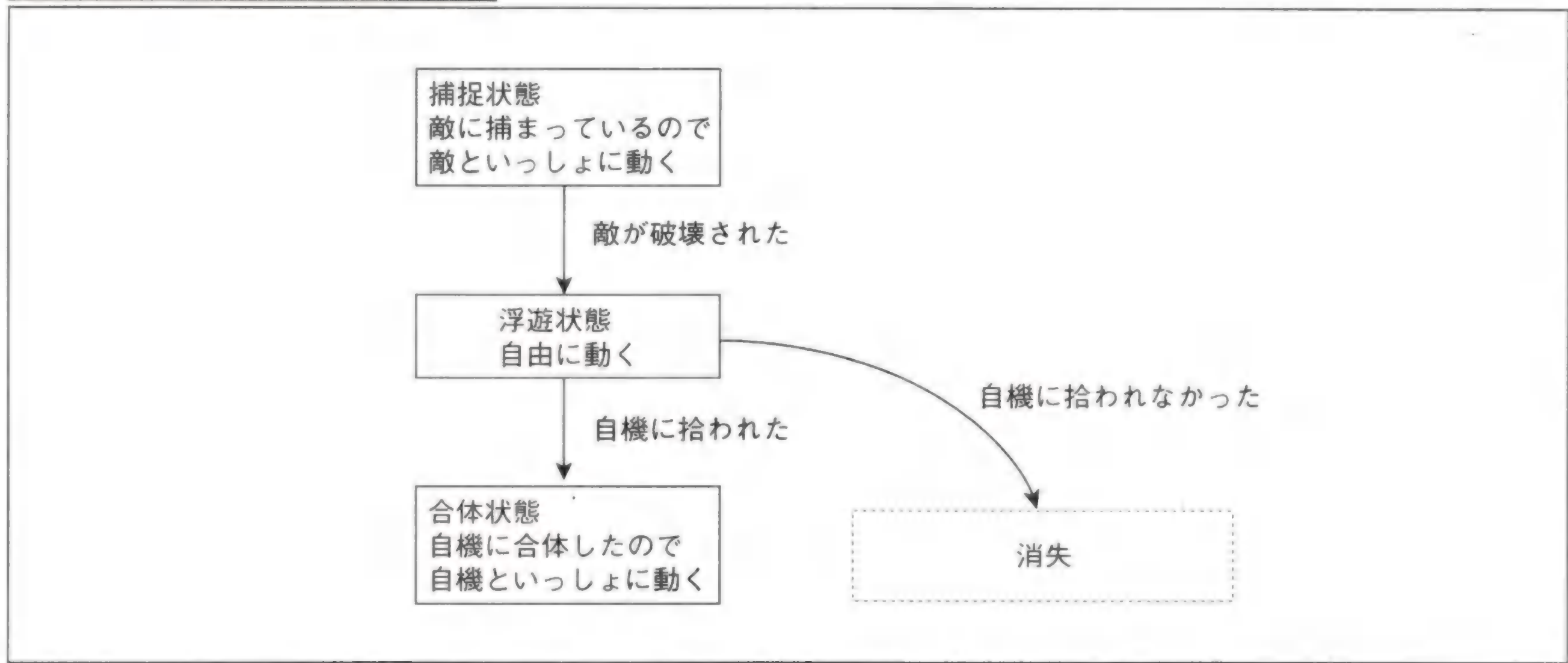
①敵を破壊することによって、敵が輸送している味方（またはパーツ）を奪還する
②自由になった味方は浮遊する
③自機で味方を拾うと合体して、自機がパワーアップする

この手順でキーとなるのは味方の状態です（Fig. 3-22）。味方には「捕捉」「浮遊」「合体」という3つの状態があり、敵が破壊されたり自機に拾われたりすることによって状態が変化します。

List 3-10は合体する味方の動きに関するプログラムです。

サンプル

合体する自機 → P. 315

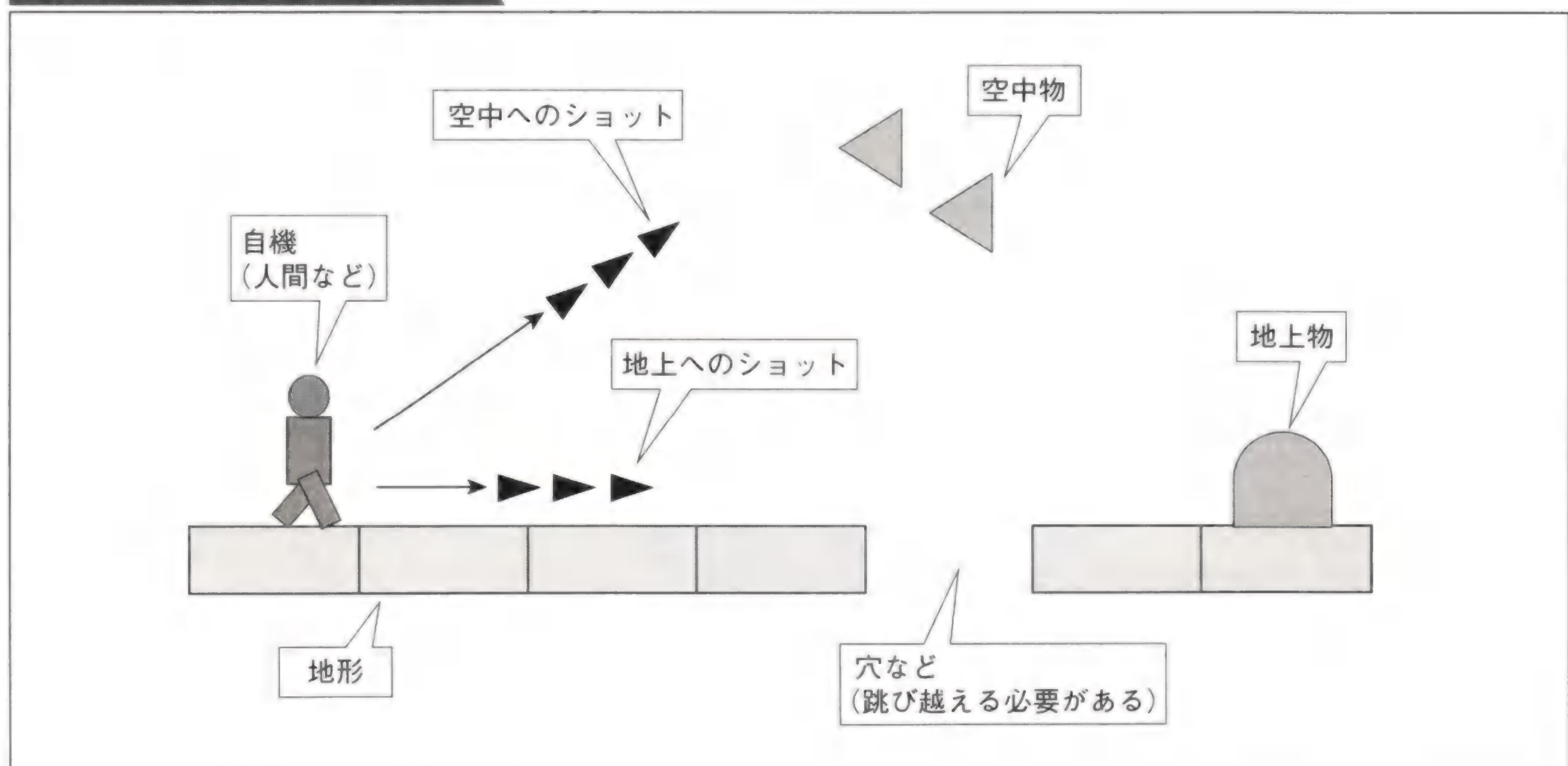
Fig. 3-22 味方の状態遷移

List 3-10 合体する味方の動き

```
// 味方の状態(捕捉、浮遊、合体)
typedef enum {
    CAPTURED, FLOATING, DOCKED
} DOCKING_STATE;

// 合体に関する処理
void Docking(
    float& cx, float& cy,  // 味方の座標
    float ex, float ey,    // 敵の座標
    float mx, float my,    // 自機の座標
    DOCKING_STATE& state   // 味方の状態
) {
    // 味方の状態によって分岐する
    switch (state) {

        // 捕捉:
        // 敵といっしょに動く。
        // 敵が破壊されたら浮遊状態に移行する。
        // 敵の破壊判定処理はEnemyDestroyed関数で行うとする。
        case CAPTURED:
            cx=ex; cy=ey;
            if (EnemyDestroyed()) state=FLOATING;
            break;

        // 浮遊:
        // 自由に動く。
        // ここでは画面下方向に直進するとする。
        // 自機に接触したら合体状態に移行する。
```

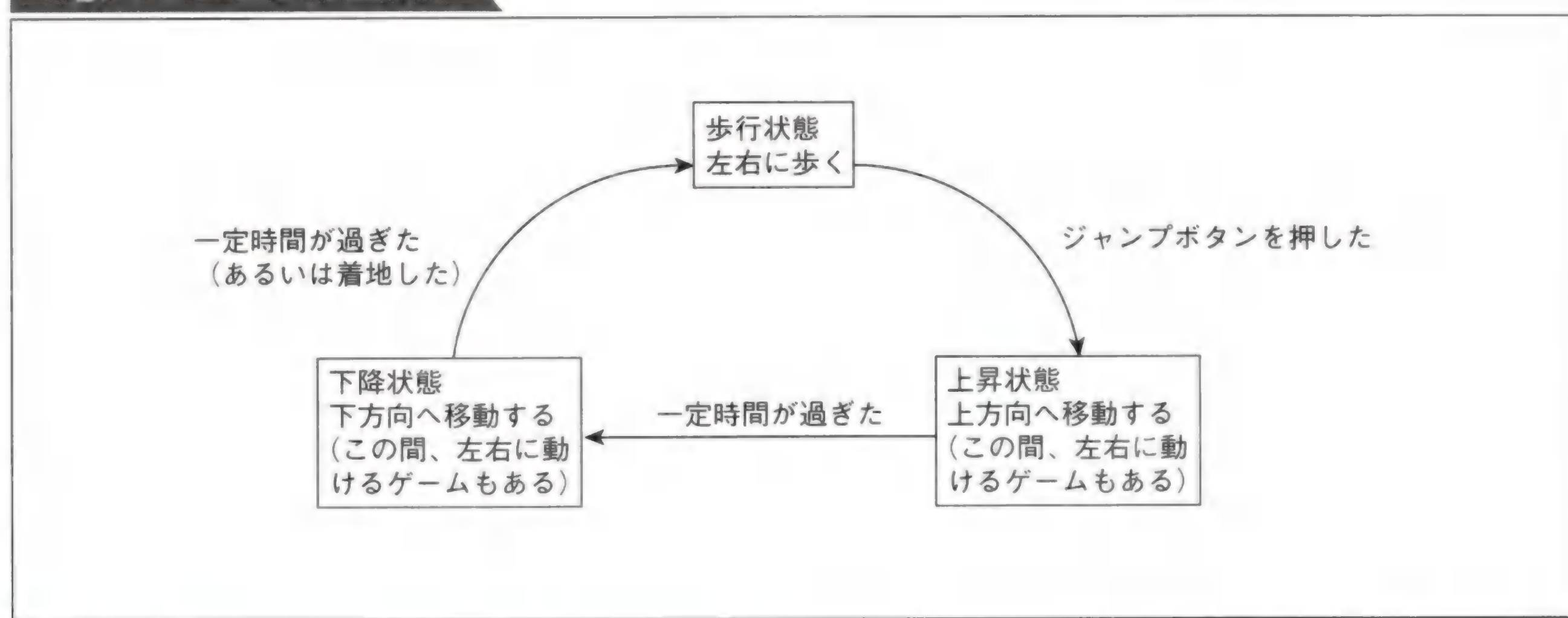
```
        // 接触判定処理はCatched関数で行うとする。
        case FLOATING:
            cy++;
            if (Catched()) state=DOCKED;
            break;

        // 合体:
        // 自機といっしょに動く。
        case DOCKED:
            cx=mx; cy=my;
            break;
    }
}
```

地上を歩く自機

人間のように地上を歩く自機です(Fig. 3-23)。シューティングゲーム全体で見ると、飛行機や宇宙船のように空中を飛行する自機のほうが多いのですが、地上を歩く自機もけっこうあります。たとえば「チェルノブ(→ P. 330)」や「メタルスラッグ(→ P. 334)」などの自機は人間です。こういったゲームは、内容によってシューティングゲームに分類されることもあれば、アクションゲームに分類されることもあります。

Fig. 3-23 地上を歩く自機

地上を歩く自機は、左右方向には自由に動けますが、上下方向には自由に動けません（横スクロールの場合）。上下に動くにはジャンプするか、はしごを上り下りするなどの手段を使います。地上には穴やバリケードなどの障害物が出現するので、そういった障害物はジャンプで跳び越えたり、あるいはショットで破壊したりします。また、空中に敵が出現するゲームでは、自機のショット方向をスティック操作で変えられることがあります。

地上を歩く自機の動きはゲームによって違いますが、ジャンプによるシンプルな自機の動きはFig. 3-24のようになります。歩行状態でジャンプボタンを押すとジャンプして上昇します。一定時間だけ上昇すると下降に移り、また一定時間が経ったところで歩行状態に戻ります。地面に起伏があるゲームの場合には、自機の足下に地面があるかどうかを調べて、地面がないときには落下させる、という処理を行います。

Fig. 3-24 自機の状態遷移

List 3-11は地上を歩く自機の動きをまとめたプログラムです。ここではジャンプ中も左右へ自由に動けるようにしました。ジャンプは一定速度になっていますが、加速度（重力加速度）をつけても面白いでしょう。

サンプル

地上を歩く自機 → P. 315

List 3-11 地上を歩く自機の動き

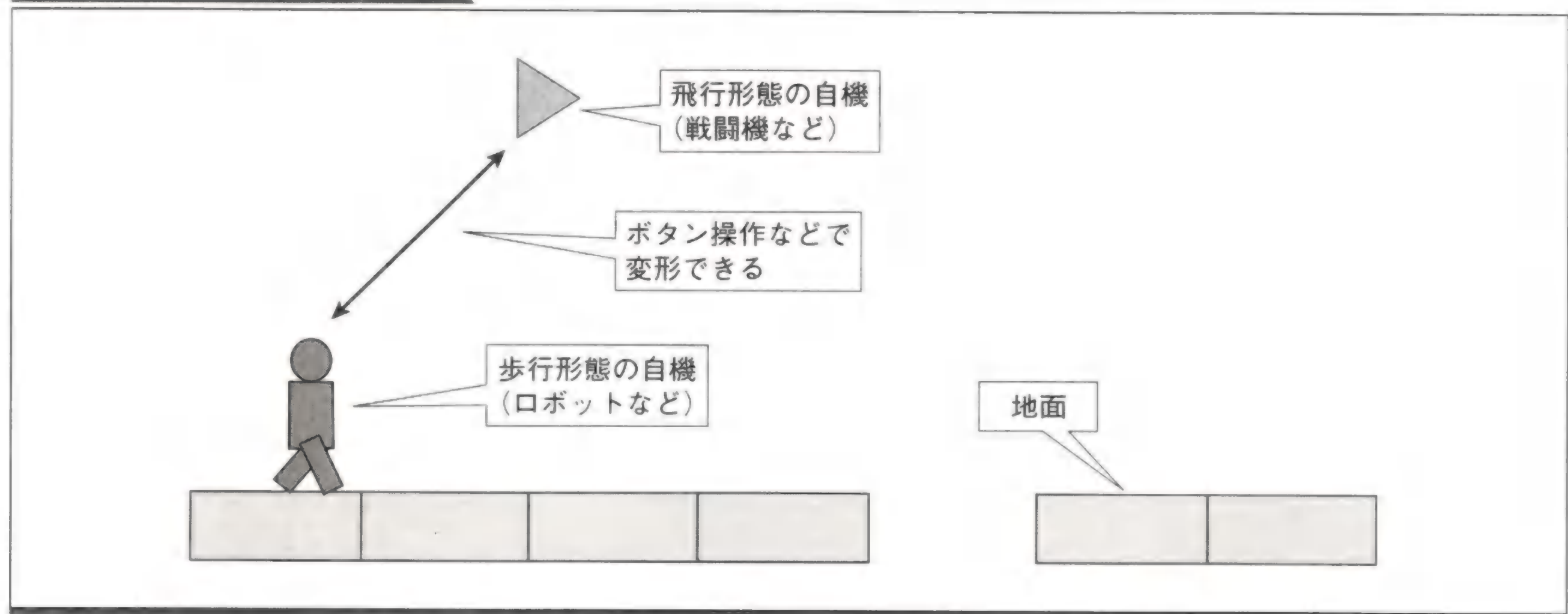
```
// 自機の状態（歩行、上昇、下降）
typedef enum {
    WALKING, JUMP_UP, JUMP_DOWN
} MYSHIP_STATE;

// 自機の動き
```

```
void Walk(
    float& x, float& y,      // 自機の座標
    float speed,             // 自機の速さ
    bool left, bool right,   // スティック入力(左右)
    bool jump_button         // ジャンプボタンの入力
) {
    static MYSHIP_STATE state=WALKING;  // 自機の状態
    static int time;                    // ジャンプの時間

    // 左右への移動
    if (left ) x-=speed;
    if (right) x+=speed;

    // 状態によって分岐する
    switch (state) {

        // 歩行:
        // ジャンプボタンを押したら上昇状態に移行する。
        case WALKING:
            if (jump_button) {
                state=JUMP_UP;
                time=40;
            }
            break;

        // 上昇:
        // 一定時間上昇したら下降状態に移行する。
        case JUMP_UP:
            y-=speed;
            if (time==0) {
                state=JUMP_DOWN;
                time=40;
            } else time--;
            break;

        // 下降:
        // 一定時間下降したら歩行状態に戻る。
        // 一定時間下降させるかわりに、
        // 自機の足下を調べて
        // 地面がないときに落下させる方法もある。
        case JUMP_DOWN:
            y+=speed;
            if (time==0) state=WALKING; else time--;
            break;
    }
}
```

変形する自機

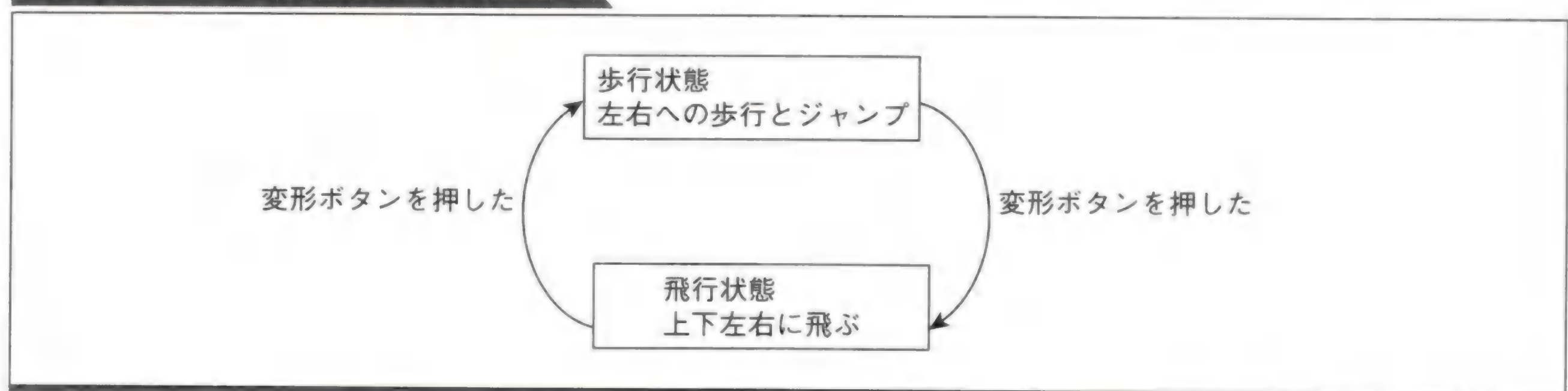
自機の形態を変えることによって、移動や攻撃などの特性が変化するという要素です（Fig. 3-25）。このルールを採用しているゲームはそれほど多くはありませんが、たとえば「フォーメーションZ（→ P. 333）」では自機の形態を歩行形態（ロボット）と飛行形態（戦闘機）とに切り替えることができます。「超時空要塞マクロス（→ P. 330）」でも自機の形態を3種類に切り替えることができますが、こちらはアイテムを取ることによって自機が変形するものなので、どちらかといえばパワーアップや武器切り替えに近いものです。

Fig. 3-25 変形する自機

飛行形態の自機
（戦闘機など）
ボタン操作などで
変形できる
歩行形態の自機
（ロボットなど）
地面

自機が変形する場合には、自機の形態によって移動や攻撃の処理を切り替えます（Fig. 3-26）。自機の変形には変形ボタンを使います。

List 3-12は変形する自機の処理をまとめたプログラムです。自機の移動に関する具体的な処理は、「自機の移動」（→ P. 74）や「地上を歩く自機」（→ P. 93）で解説しています。

Fig. 3-26 変形する自機の状態遷移

歩行状態
左右への歩行とジャンプ
変形ボタンを押した
変形ボタンを押した
飛行状態
上下左右に飛ぶ

サンプル

変形する自機 → P. 315

List 3-12 変形する自機

```
// 自機の形態（歩行、飛行）
typedef enum {
    WALKING, FLYING
} MYSHIP_FORM;

// 自機の変形
void Transform(
    bool button  // 変形ボタンの入力
) {
    static MYSHIP_FORM formation=WALKING;  // 自機の形態
    static bool prev_button=false;         // 前回のボタン入力

    // 変形：
    // 変形ボタンが押されたら変形する。
    if (!prev_button && button) {
        if (formation==WALKING) formation=FLYING;
            else formation=WALKING;
    }
    prev_button=button;

    // 形態によって動きを変える：
    // 具体的な処理はWalk、Flyの各関数で行うとする。
    if (formation==WALKING) Walk(); else Fly();
}
```

水中の移動

ゲームによっては自機が水中を移動することがあります（Fig. 3-27）。そのようなゲームの多くは、自機が水中に入ったときには少し動きが鈍くなります。

水面の高さが一定のゲームでは、水中に入ったかどうかは自機の高度を調べればわかります（Fig. 3-28）。自機の高度（Y座標）が水面より高ければ空中、水面よりも低ければ水中というわけです。

水面の高さが一定ではないゲームの場合には、水との当たり判定処理を行うことによって、自機が水中にいるかどうかを判定します（Fig. 3-29）。

Fig. 3-27 水中の移動

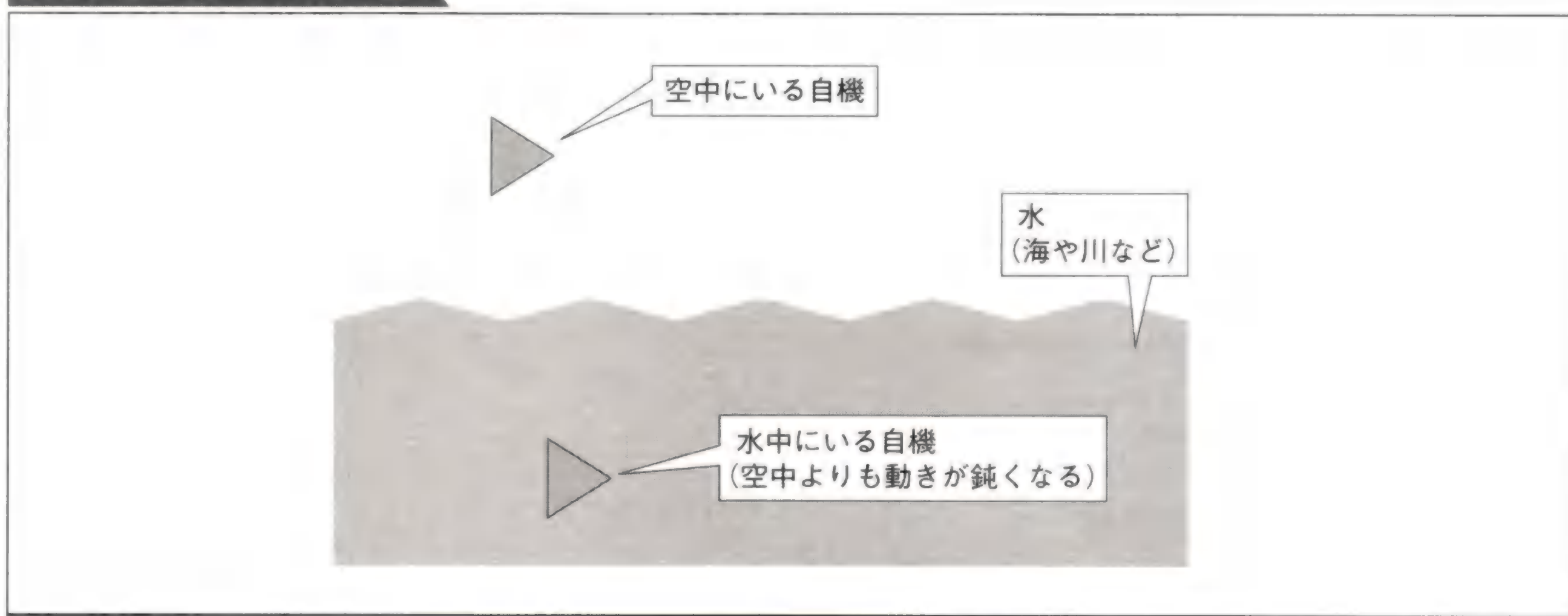



Fig. 3-28 自機の高度を調べる

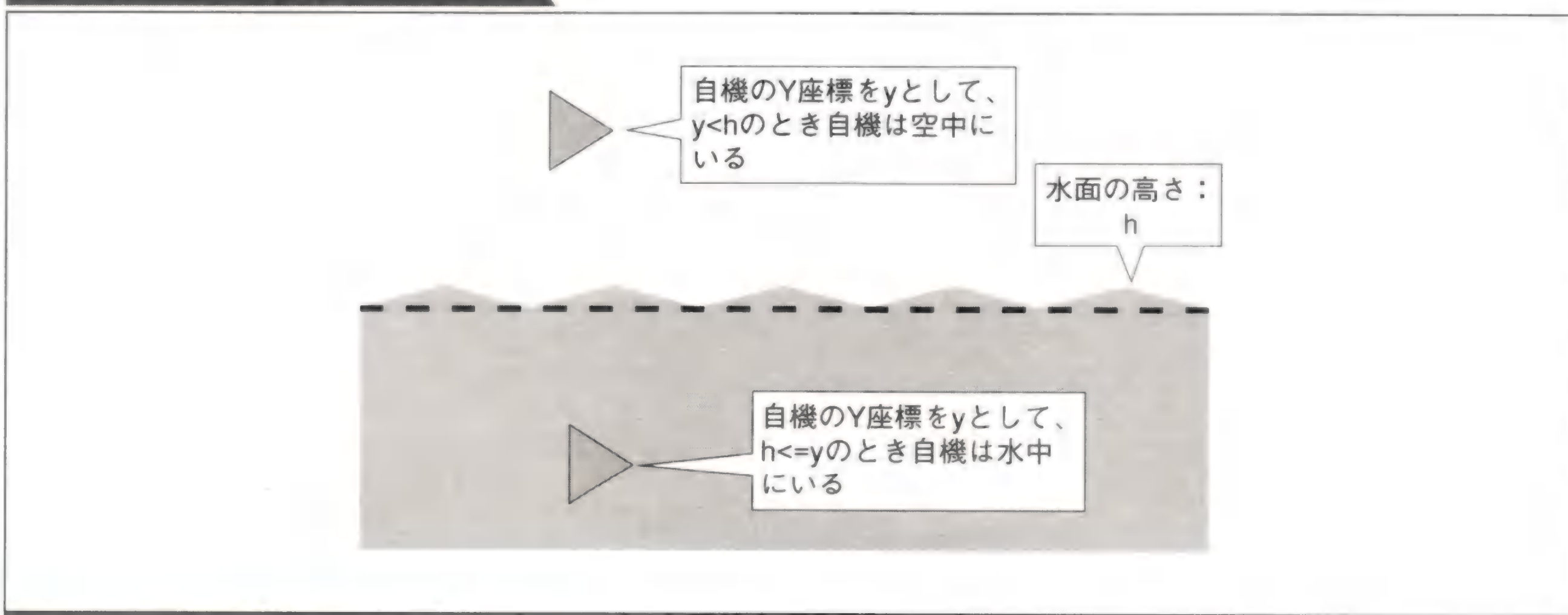
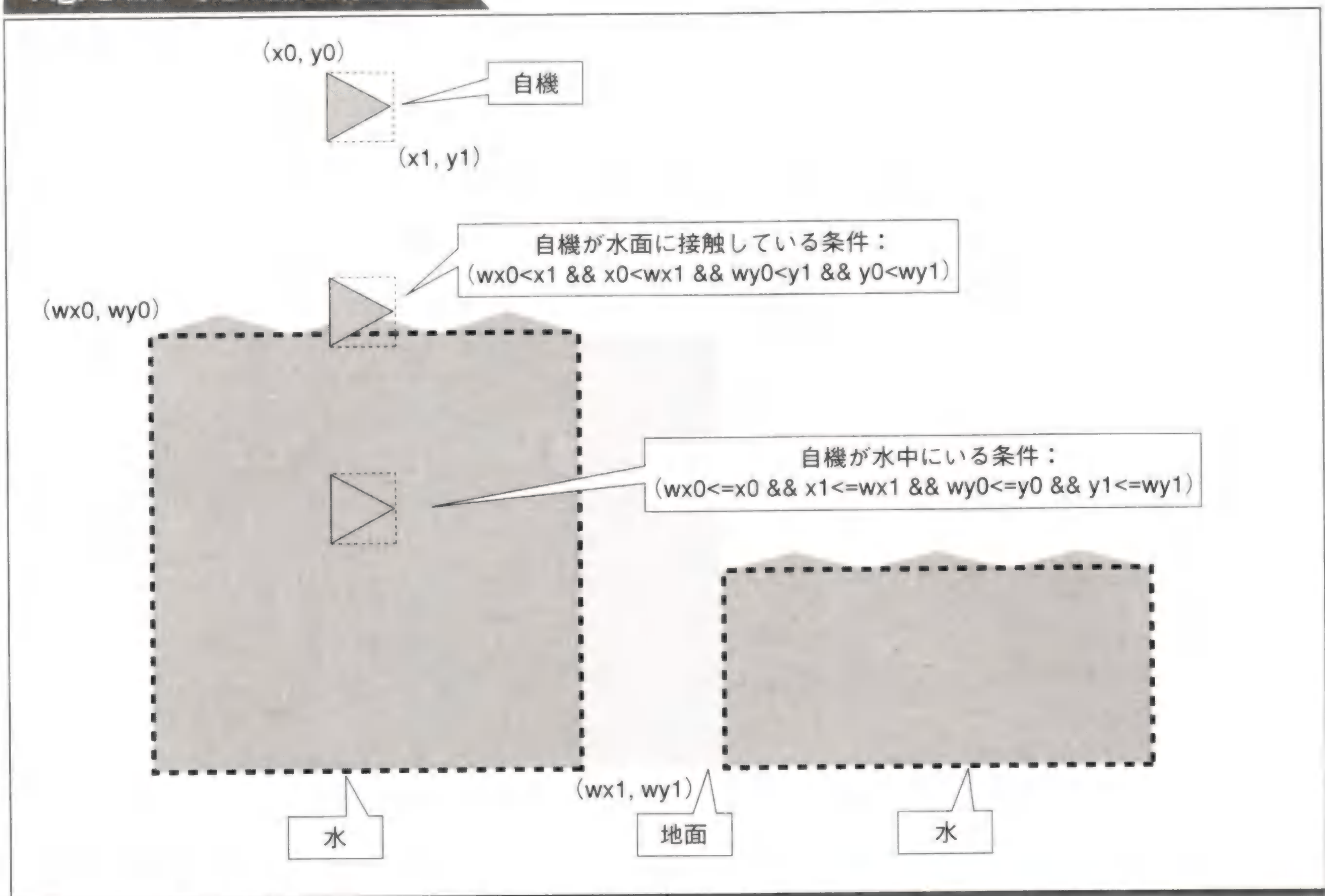



Fig. 3-29 水との当たり判定処理

画面中のすべての水を一種の「地形」と見なして、すべての水に当たり判定を設定しておきます。自機が水にすっぽり入っていれば、自機は水中にいるというわけです。自機の左上座標を (x0, y0)、右下座標を (x1, y1)、水の当たり判定の左上座標を (wx0, wy0)、右下座標を (wx1, wy1) とすると、自機が水中にいる条件は次のようになります。

(wx0<=x0 && x1<=wx1 && wy0<=y0 && y1<=wy1)

また、自機が水面に接触している条件は次のようになります。

(wx0<x1 && x0<wx1 && wy0<y1 && y0<wy1)

自機が水面に接触しているときには、波しぶきなどのエフェクトを表示すると雰囲気が出ます。

List 3-13は自機の高度を調べることによって、水中かどうかを判定するプログラムです。水中の場合には移動速度を空中の半分に落とします。List 3-14では、水との当たり判定処理を行うことによって、水中かどうかを判定します。自機が水面に接触しているかどうかも調べ、接触している場合には波しぶきなどのエフェクトを表示します。

サンプル

水中の移動 → P. 316

List 3-13 水中の移動(自機の高度を調べる場合)

```
void Underwater1(
    float& x, float& y,  // 自機の座標
    float vx, float vy,  // 自機の速度
    float h              // 水面の高さ
) {
    // 空中の場合：
    // 通常の速度で移動する。
    if (y<h) {
        x+=vx; y+=vy;
    }

    // 水中の場合：
    // 速度を遅くする。ここでは空中の半分とした。
    else {
        x+=vx/2; y+=vy/2;
    }
}
```

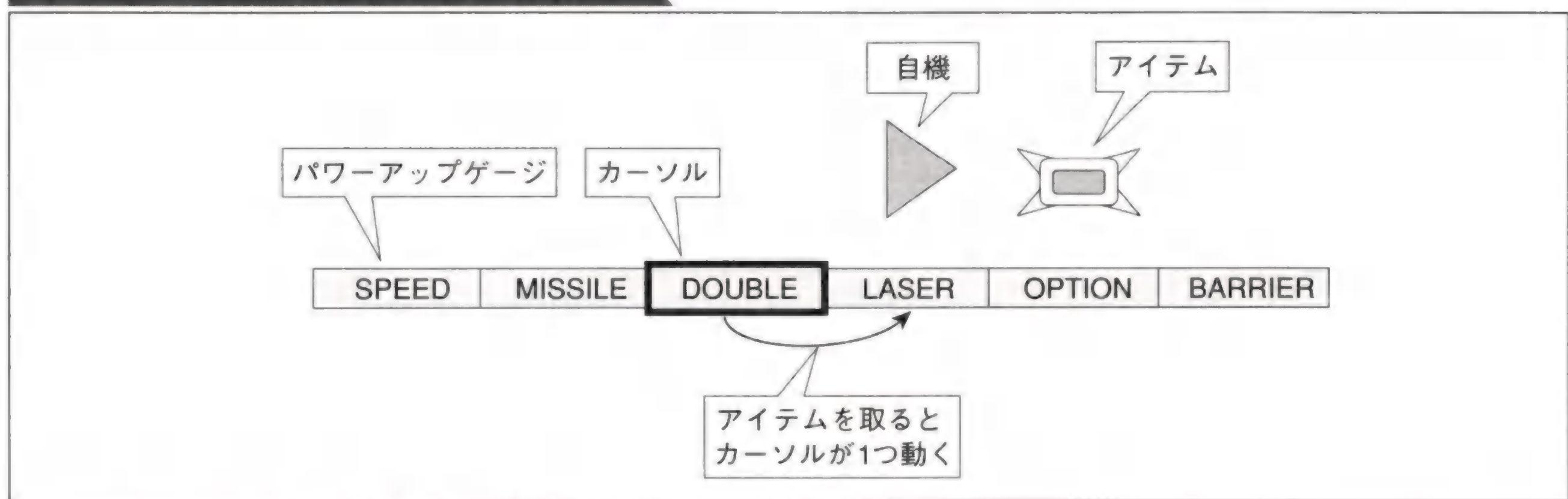
List 3-14 水中の移動(水との当たり判定処理を行う場合)

```
void Underwater2(
    float& x, float& y,          // 自機の座標
    float x0, float y0,          // 自機の左上座標
    float x1, float y1,          // 自機の右下座標
    float vx, float vy,          // 自機の速度
    float wx0[], float wy0[],    // 水の当たり判定
    float wx1[], float wy1[],    // (左上座標、右下座標)
    int num_water                // 水の当たり判定の数
) {
    // 自機が水中にいるかどうかを調べる
    int i;
    for (i=0; i<num_water; i++) {
        if (wx0[i]<=x0 && x1<=wx1[i] &&
            wy0[i]<=y0 && y1<=wy1[i]) break;
    }

    // 自機が水中にいる場合：
    // 速度を遅くする。ここでは空中の半分とした。
    if (i<num_water) {
        x+=vx/2; y+=vy/2;
```

```
    }

    // 自機が水中にはいない場合：
    // 通常の速度で移動する。
    else {
        x+=vx; y+=vy;

        // 自機が水面に接触しているかどうかを調べる
        for (i=0; i<num_water; i++) {
            if (wx0[i]<x1 && x0<wx1[i] &&
                wy0[i]<y1 && y0<wy1[i]) break;
        }

        // 自機が水面に接触している場合：
        // 波しぶきなどのエフェクトを表示する。
        // 具体的な処理はWaveEffect関数で行うとする。
        if (i<num_water) WaveEffect();
    }
}
```

ゲージを使ったパワーアップ

パワーアップアイテムを取るたびにパワーアップゲージ上のカーソルが移動し、パワーアップボタンを押すとカーソルが指している内容のパワーアップが行われるという方式です（Fig. 3-30）。この方式は「グラディウス（→ P. 326）」シリーズや「パロディウス（→ P. 332）」シリーズで採用されています。この方式のポイントは、「まずスピードを上げてからミサイルを取ろう」といった具合に、パワーアップの順番をプレイヤーが選べることです。

Fig. 3-30 ゲージを使ったパワーアップ

グラディウス方式のパワーアップゲージでは、右端にカーソルがあるときにアイテムを取ると、カーソルは左端に移動します(Fig. 3-31)。また、最初はカーソルが表示されておらず、アイテムを1つ取ると初めてカーソルが表示されます(Fig. 3-32)。

List 3-15はゲージを使ったパワーアップに関するプログラムです。たとえば、パワーアップの種類が6種類あるときには、カーソルの位置は6か所+1か所(表示されていない状態)となり、合計で7つの状態があります。

Fig. 3-31 右端にカーソルがあるとき

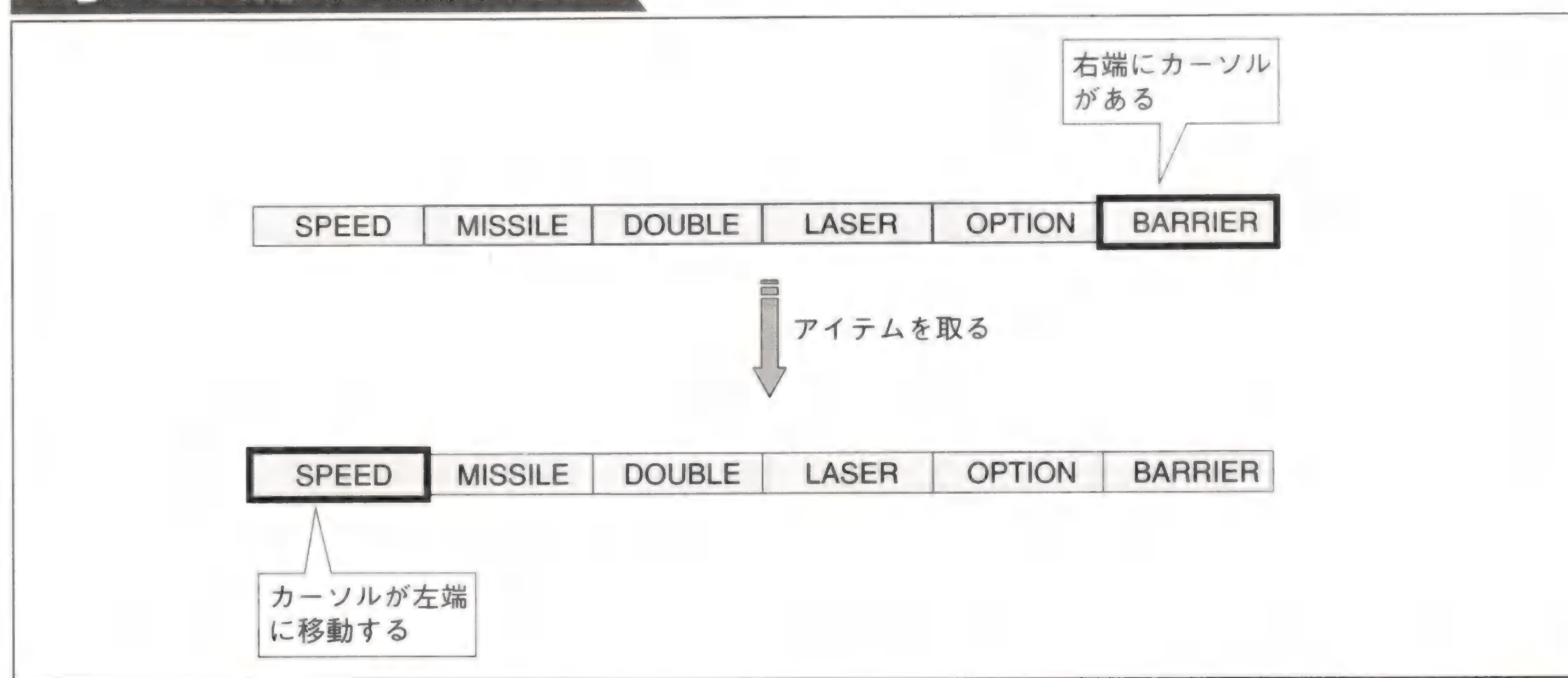
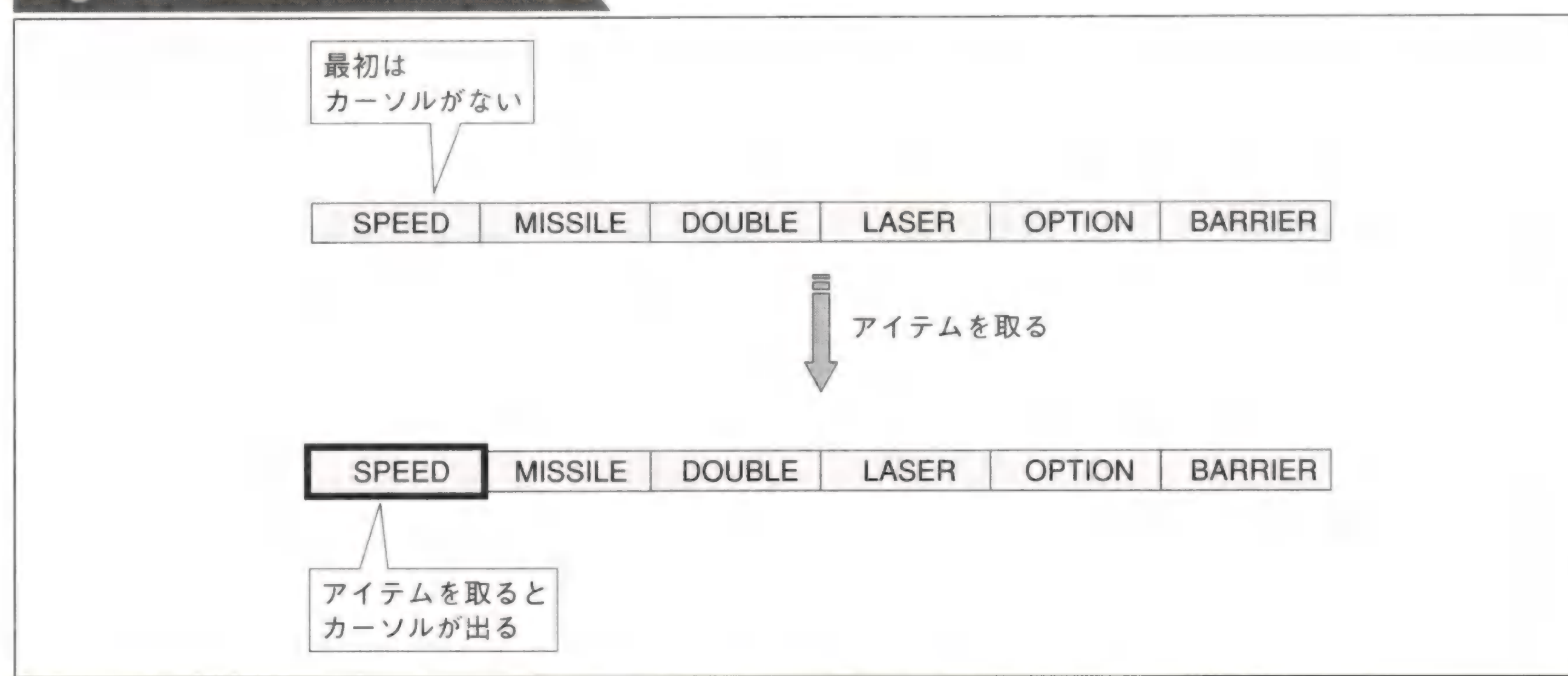



Fig. 3-32 アイテムを取る前の状態

List 3-15 ゲージを使ったパワーアップ

```
// パワーアップの種類
typedef enum {
    NONE,  // カーソルがない状態
    SPEED, MISSILE, DOUBLE,
    LASER, OPTION, BARRIER,
    END    // ゲージの右端を示す
} POWER_UP_TYPE;

// ゲージを使ったパワーアップ
void PowerUpGauge(
    bool power_up_button  // パワーアップボタンの入力
) {
    static POWER_UP_TYPE cursor=NONE;  // カーソルの位置

    // アイテムを拾ったときの処理：
    // アイテムを拾ったら、カーソルを1つ右に動かす。
    // 右端まで動いたら左端に戻す。
    // 拾い判定処理はPickItem関数で行うとする。
    if (PickItem()) {
        cursor++;
        if (cursor==END) cursor=SPEED;
    }

    // パワーアップの処理：
    // ボタンが押されたら、
    // カーソルが指しているパワーアップを行う
    // 具体的な処理はPowerUp関数で行うとする
    if (power_up_button) {
        PowerUp(cursor);
    }
}
```

オプション

オプションというのはもともとは「グラディウス（→ P. 326）」で採用されたもので、自機の航跡をトレースする球状のエネルギー体のことです（Fig. 3-33）。「グラディウス」はとても有名なゲームなので、「いわゆるオプション」とか「グラディウスのオプション」といえば通じるゲーマーやゲームプログラマは多いでしょう（年代にもよりますが）。「グラディウス」ではオプションを4つまでつけることができます。

基本的に、オプションは自機の航跡をトレースします（Fig. 3-34）。つまり、自機が描いた軌跡と同じ軌跡を描きながら、自機よりも少しだけ遅れて移動します。多くのオプションをつけていると、さながら自機に尾が生えたような状態になります。

Fig. 3-33 オプション

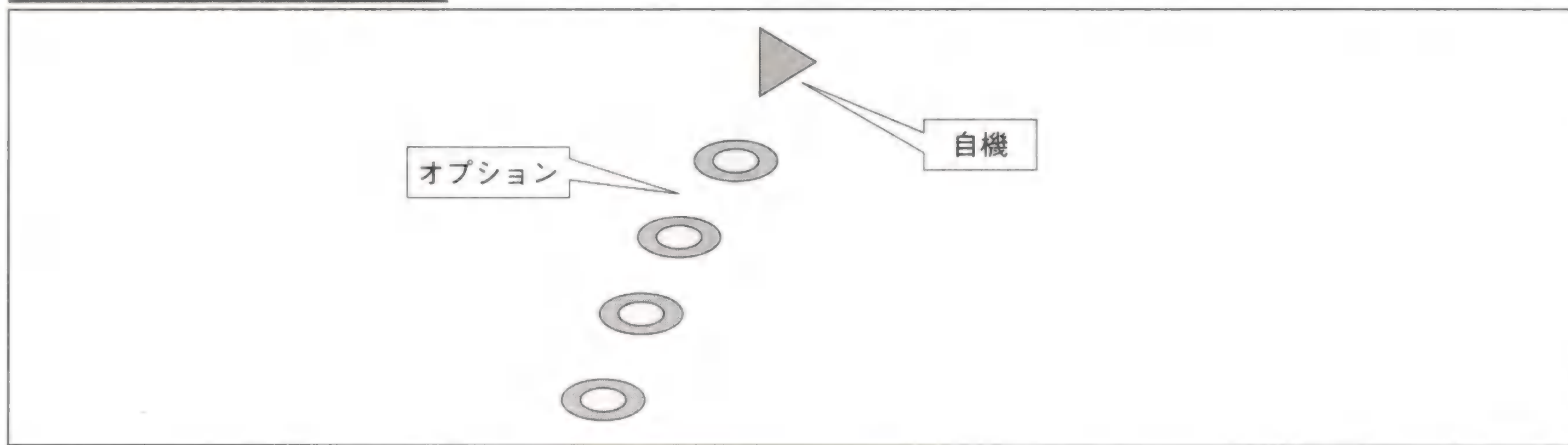
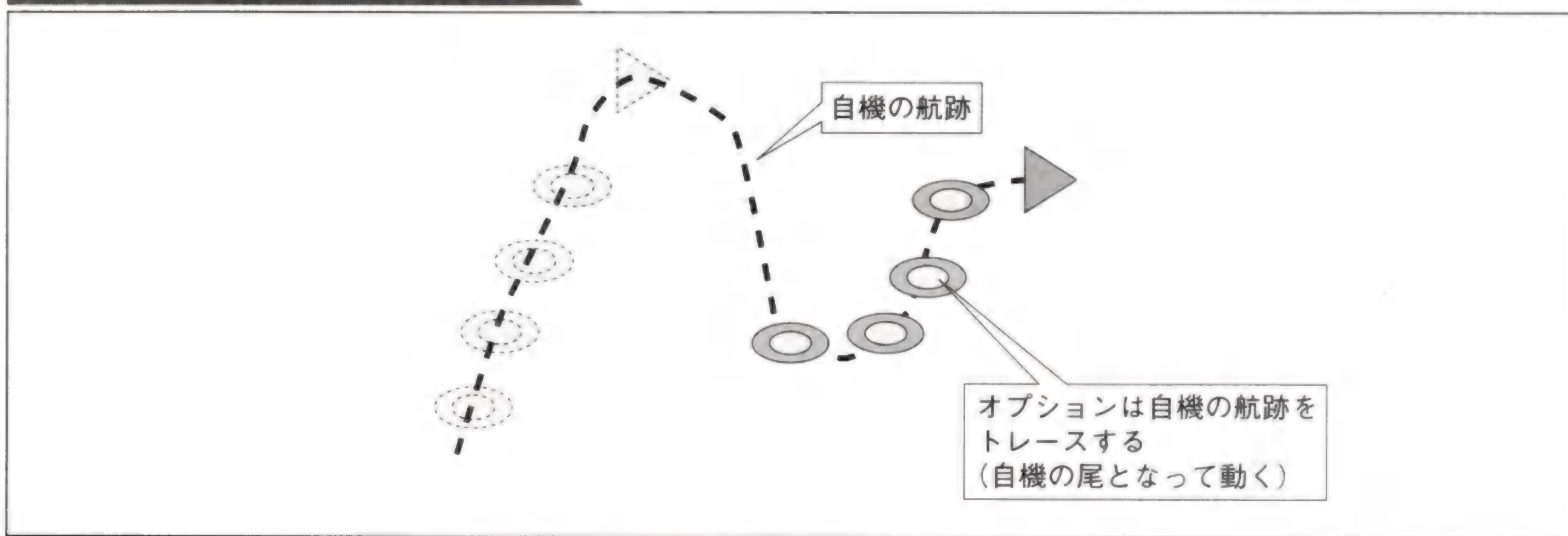



Fig. 3-34 オプションの動きかた

オプションのアルゴリズム

オプションを作るときのポイントは、過去の自機の座標を配列などに保存しておくことです。オプションは自機の航跡をトレースしますが、これは表現を変えれば「自機の過去の座標にオプションを置く（描画する）」ということです（Fig. 3-35）。

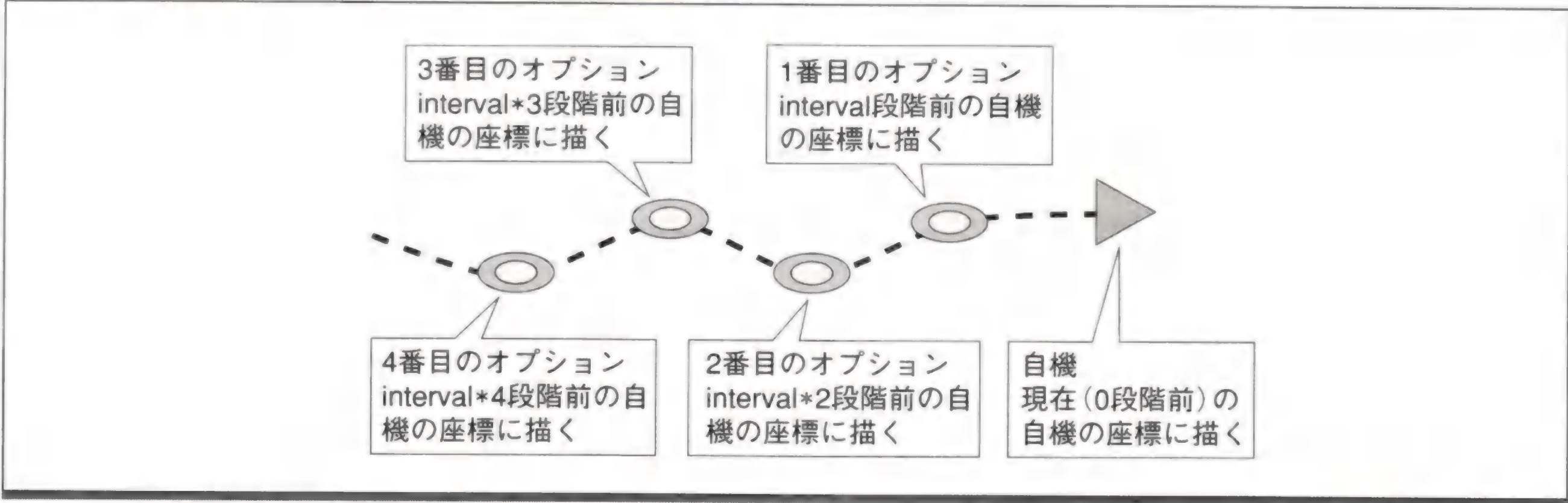
Fig. 3-35 自機の過去の座標とオプションの位置

3番目のオプション
interval*3段階前の自機の座標に描く

1番目のオプション
interval段階前の自機の座標に描く

4番目のオプション
interval*4段階前の自機の座標に描く

2番目のオプション
interval*2段階前の自機の座標に描く

自機
現在(0段階前)の自機の座標に描く

シューティングゲームでは、細かい時間間隔(1/60秒など)で自機や弾などを少しずつ動かすことを繰り返します。オプションを作るには、何段階か過去の時点における自機の座標を保存しておき、その座標にオプションを描きます。オプションの間隔をintervalで表すと、各オプションを描く際に使う自機の座標は次のようになります。

- **1番目のオプション:interval段階前の座標**
- **2番目のオプション:interval*2段階前の座標**
- **3番目のオプション:interval*3段階前の座標**

……(以下同様)………

自機から遠いオプションほど、より古い自機の座標を使うわけです。

Fig. 3-36 自機の座標の遷移

①自機の座標を位置index(0)に書き込み、indexを+1する

0(index)	1	2	3	4	5(length-1)
0段階前の座標(自機の座標)	5段階前の座標	4段階前の座標(2番目のオプション)	3段階前の座標	2段階前の座標(1番目のオプション)	1段階前の座標

②自機の座標を位置index(1)に書き込み、indexを+1する

0	1(index)	2	3	4	5(length-1)
1段階前の座標	0段階前の座標(自機の座標)	5段階前の座標	4段階前の座標(2番目のオプション)	3段階前の座標	2段階前の座標(1番目のオプション)

③ indexがlength-1に達したら、次のindexは0に戻す(①の状態に戻る)

0	1	2	3	4	5(index, length-1)
5段階前の座標	4段階前の座標(2番目のオプション)	3段階前の座標	2段階前の座標(1番目のオプション)	1段階前の座標	0段階前の座標(自機の座標)

④最初は配列の全要素に自機の座標を書き込んでおく

0(index)	1	2	3	4	5(length-1)
自機の座標	自機の座標	自機の座標	自機の座標	自機の座標	自機の座標

自機の座標を保存するには配列を使います（Fig. 3-36）。ここでは現在の自機の座標を書き込む位置（配列のインデクス）をindexとし、座標の個数（配列の長さ）をlengthとします。最初はindexを0にしておきます。Fig. 3-36ではlengthを6としました。

indexが0のとき、自機の座標は配列の0番目の要素に保存します。このとき、オプションの間隔を2とすると、1番目のオプションの座標は自機の2段階前の座標、つまり配列の4番目の要素になります。また、2番目のオプションの座標は自機の4（=2*2）段階前の座標、つまり配列の2番目の要素になります（Fig. 3-36-①）。

自機の座標を保存したらindexを+1して、次は配列の1番目の要素に自機の座標を保存します（Fig. 3-36-②）。このとき、オプションの座標は配列の5番目と3番目の要素になります。

配列の長さは有限なので、自機の座標を次々に保存していくと、そのうちにindexが配列の末尾（length−1番目の要素）に到達してしまいます（Fig. 3-36-③）。このように配列の末尾に達したら、次はindexを0に戻して、①の状態に戻します。

なお、自機が画面に出現した瞬間（ゲームの開始時やステージの開始時など）には、自機はまだ動いていないので、配列に自機の過去の座標が保存されていません。これではオプションを描くのに不都合なので、最初は配列の全要素に同じ座標を書き込んでおきます（Fig. 3-36-④）。

座標を保存するために必要な配列の長さlengthは、オプションの間隔をinterval、オプションの個数をcountとすると、次のように求められます。

length=interval*count

たとえばintervalを10、countを4とすると、長さ10*4=40の配列が必要だということです。

■ オプションのプログラム

List 3-16はオプションの初期化処理と移動処理をまとめたものです。InitOptionは初期化処理で、自機が画面に出現したときに1回だけ呼び出します。MoveOptionはオプションを動かすたびに呼び出します。

List 3-16のなかにある、

```
i= (i-opt_interval+length) %length;
index= (index+1) %length;
```

という2つの式では、剰余演算（%）を使って配列の末尾に達したときに先頭に戻る処理を行います。1番目の式でlengthを足しているのは、i-opt_intervalが負の数になったときに、これを正の数に戻すためです。負の数になると剰余の結果が狂ってしまいます。

なお、List 3-16のオプションは、自機が動いていないときでも動くオプションです。自機を停止させると、尻尾が縮むようにオプションが自機に近づいてきます。このように縮まないオプションを作るには、自機が停止しているときに自機の座標を保存しないようにします。

サンプル

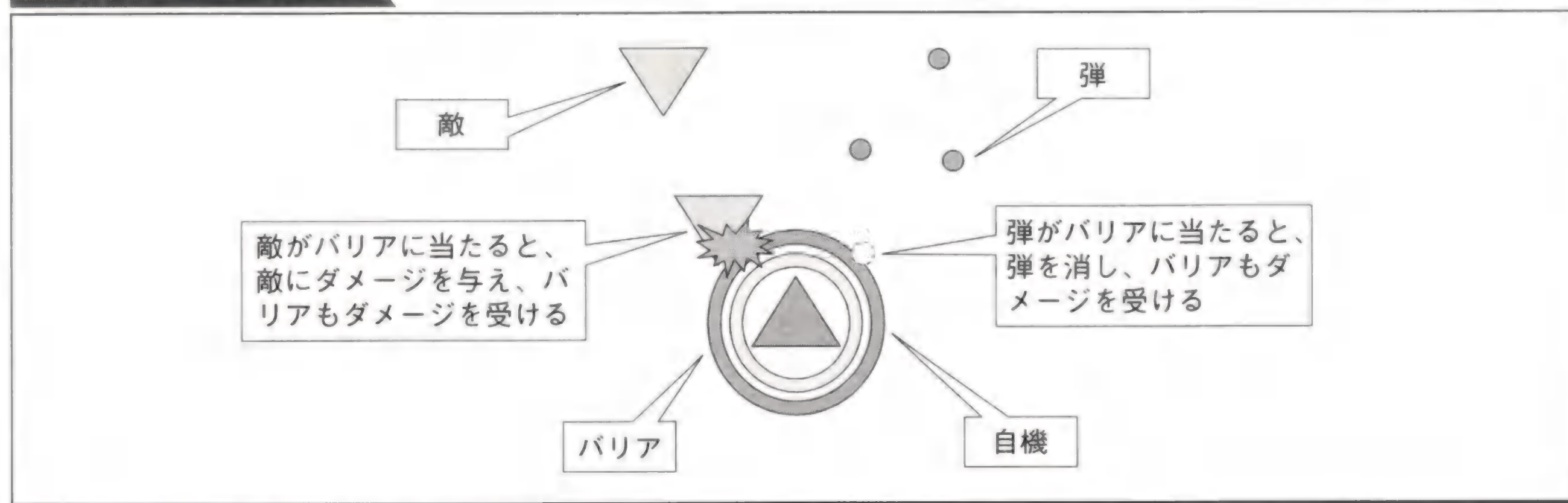
オプション →P. 316

List 3-16 オプションの初期化と移動

```
// オプションの初期化
void InitOption(
    float x, float y,         // 自機の座標（X方向、Y方向）
    float ox[], float oy[],   // 自機の古い座標（配列）
    int length                // 古い座標の個数（配列の長さ）
) {
    // 配列の全要素を自機の座標で初期化する
    for (int i=0; i<length; i++) {
        ox[i]=x; oy[i]=y;
    }
}

// オプションの移動
void MoveOption(
    float x, float y,         // 自機の座標（X方向、Y方向）
    float ox[], float oy[],   // 自機の古い座標（配列）
    int length,               // 古い座標の数（配列の長さ）
    int& index,               // 座標の保存位置（配列上の位置）
    int opt_count,            // オプションの個数
    int opt_interval          // オプションの間隔（配列上の間隔）
) {
    // オプションを描く：
    // DrawOptionはオプションを描く関数とする。
    for (int c=0, i=index; c<opt_count; c++) {
        i=(i-opt_interval+length)%length;
        DrawOption(ox[i], oy[i]);
    }

    // 自機の座標を保存し、座標の保存位置を更新する
    ox[index]=x;
    oy[index]=y;
    index=(index+1)%length;
}
```

バリア

自機の周りに展開されて、自機を守る壁です（Fig. 3-37）。バリアを張る方法はゲームによって違いますが、多くのゲームでは、アイテムを取ったりパワーアップしたりすることによってバリアを張ります。

Fig. 3-37 バリア

バリアの効果はゲームによってさまざまですが、次のようなものが一般的です。

- **敵がバリアに当たると、敵にダメージを与え、バリアもダメージを受ける**
- **弾がバリアに当たると、弾を消し、バリアもダメージを受ける**

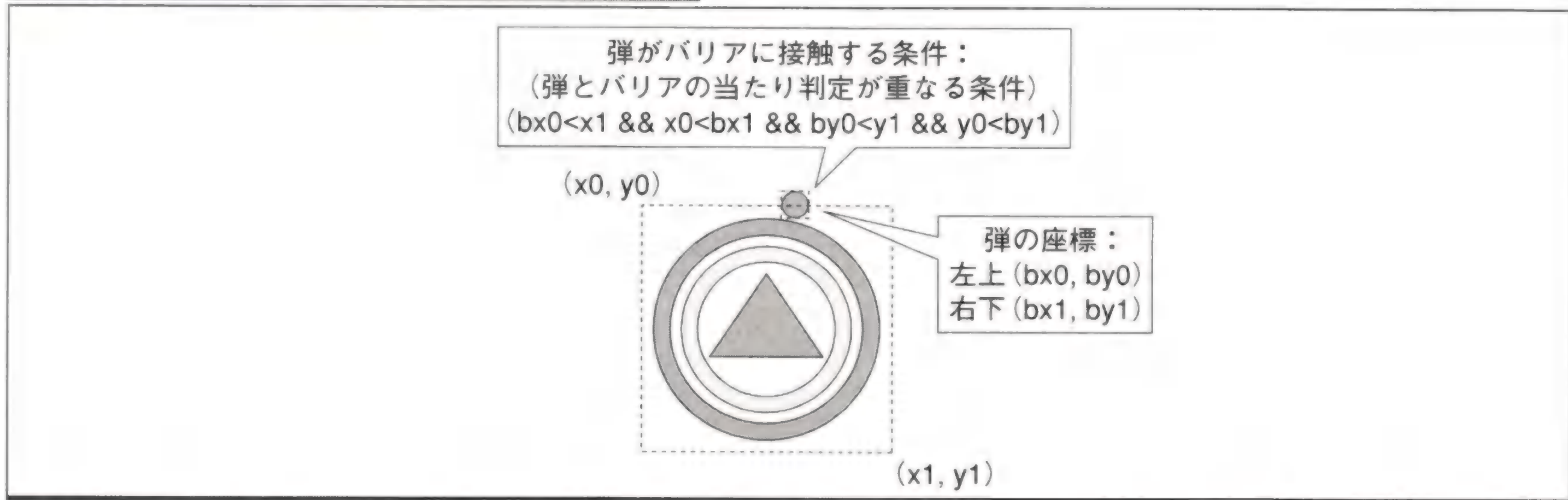
バリアに蓄積したダメージが限界値を超えると、バリアは消滅します。ダメージを受けるとバリアが小さくなったり色が変わったりするゲームもあります。

バリアに敵や弾が当たったかどうかを調べるには、当たり判定処理を行います（Fig. 3-38）。たとえばバリアの当たり判定の左上座標を（x0, y0）、右下座標を（x1, y1）、弾の当たり判定の左上座標を（bx0, by0）、右下座標を（bx1, by1）とすると、バリアに弾が当たる条件は次のようになります。

(bx0<x1 && x0<bx1 && by0<y1 && y0<by1)

敵がバリアに当たる条件も同様です。

List 3-17はバリアに関するプログラムです。ここでは弾も敵も一定のダメージ値にしていますが、弾や敵の種類によってバリアに与えるダメージの大きさを変えてもよいでしょう。

Fig. 3-38 バリアと弾との当たり判定処理

サンプル

バリア → P. 316

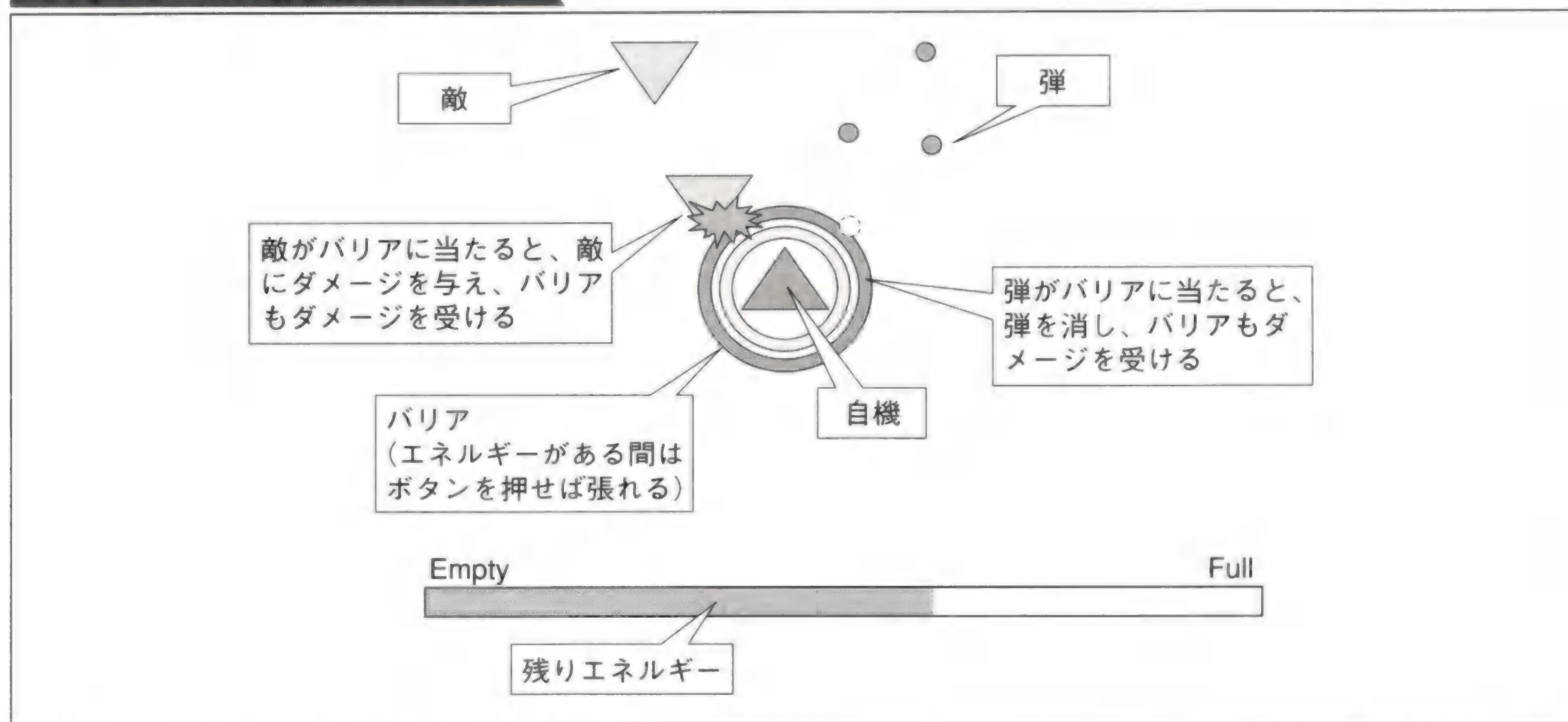
List 3-17 バリア

```
void Barrier1(
    float x0, float y0,          // バリアの当たり判定
    float x1, float y1,          // （左上座標、右下座標）
    float bx0[], float by0[],    // 弾の当たり判定
    float bx1[], float by1[],    // （左上座標、右下座標）
    int num_bullet,              // 弾の数
    float ex0[], float ey0[],    // 敵の当たり判定
    float ex1[], float ey1[],    // （左上座標、右下座標）
    int num_enemy,               // 敵の数
    int& damage,                 // バリアに蓄積されたダメージ
    int max_damage               // バリアのダメージの限界値
) {
    // バリアと弾との当たり判定処理：
    // 弾が当たったら、弾を消し、バリアのダメージを増やす。
    // 弾を消す具体的な処理はDeleteBullet関数で行うとする。
    for (int i=0; i<num_bullet && damage<max_damage; i++) {
        if (bx0[i]<x1 && x0<bx1[i] &&
            by0[i]<y1 && y0<by1[i]) {
            DeleteBullet(i);
            damage++;
        }
    }

    // バリアと敵との当たり判定処理：
    // 敵が当たったら、敵にダメージを与え、
    // バリアのダメージも増やす。
    // 敵にダメージを与える具体的な処理は、
```

```
    // DamageEnemy関数で行うとする。
    for (int i=0; i<num_enemy && damage<max_damage; i++) {
        if (ex0[i]<x1 && x0<ex1[i] &&
            ey0[i]<y1 && y0<ey1[i]) {
            DamageEnemy(i);
            damage++;
        }
    }

    // ダメージが限界値を超えたとき：
    // バリアを消滅させる。
    // 消滅の具体的な処理はDeleteBarrier関数で行うとする。
    if (damage>=max_damage) DeleteBarrier();
}
```

ボタンで張るバリア

バリアボタンを押すことで張ることができるバリアです（Fig. 3-39）。このタイプのバリアは「グロブダー（→ P. 326）」などに見られます。「エスプレイド（→ P. 324）」や「エスプガルーダ（→ P. 324）」のバリアもボタンで張ることができますが、これらは攻撃手段にもなっており、どちらかといえばボムに近い性質です。

Fig. 3-39 ボタンで張るバリア

ボタンで張るタイプのバリアは、もちろん無制限に使えるわけではなくて、バリアを張っている間はエネルギーが減っていきます。エネルギーがなくなると張っているバリアは消滅し、ボタンを押しても再びバリアを張ることはできなくなります。

バリアとしての性能は「バリア」(→ P. 108) で解説したものとほぼ同じです。ゲームによって違いはありますが、バリアに当たった敵にダメージを与えたり弾を消したりすることができます。バリアに敵や弾が当たると、バリアのエネルギーが減ります。

List 3-18はボタンで張るバリアのプログラムです。当たり判定処理は「バリア」で解説したのとほぼ同じ方法で行います。

List 3-18 ボタンで張るバリア

```
void Barrier2(
    float x0, float y0,         // バリアの当たり判定
    float x1, float y1,         // (左上座標、右下座標)
    float bx0[], float by0[],   // 弾の当たり判定
    float bx1[], float by1[],   // (左上座標、右下座標)
    int num_bullet,             // 弾の数
    float ex0[], float ey0[],   // 敵の当たり判定
    float ex1[], float ey1[],   // (左上座標、右下座標)
    int num_enemy,              // 敵の数
    int& energy,                // バリアのエネルギー
    bool barrier_button         // バリアボタンの状態
) {
    // バリアを張る:
    // バリアボタンが押されており、
    // かつエネルギーがあるときには、バリアを張る。
    if (barrier_button && energy>0) {

        // バリアと弾との当たり判定処理:
        // 弾が当たったら、弾を消し、バリアのエネルギーを減らす。
        // 弾を消す具体的な処理はDeleteBullet関数で行うとする。
        for (int i=0; i<num_bullet && energy>0; i++) {
            if (bx0[i]<x1 && x0<bx1[i] &&
                by0[i]<y1 && y0<by1[i]) {
                DeleteBullet(i);
                energy--;
            }
        }

        // バリアと敵との当たり判定処理:
        // 敵が当たったら、敵にダメージを与え、
        // バリアのエネルギーを減らす。
        // 敵にダメージを与える具体的な処理は、
        // DamageEnemy関数で行うとする。
        for (int i=0; i<num_enemy && energy>0; i++) {
            if (ex0[i]<x1 && x0<ex1[i] &&
```

```
            ey0[i]<y1 && y0<ey1[i]) {
            DamageEnemy(i);
            energy--;
        }
    }

    // エネルギーの消費
    energy--;
  }
}
```

Stage 3 のまとめ ▶▶

自機はシューティングゲームに欠かせない要素ですが、実は弾や武器に比べると地味な存在です。しかし、ゲームを遊ぶプレイヤーの目にもっとも長い時間映っているのは、おそらく自機でしょう。それだけに、自機のデザインや動きが魅力的かどうかは、ゲーム全体の魅力にかかわってきます。

複数種類の自機を用意するのもよい工夫です。プレイヤーは自分のプレイスタイルに合わせて気に入った自機を選ぶことができますし、1つの自機でクリアしたあともほかの自機によるプレイを楽しむことができます。作る側にとっては自機の種類が増えるぶん、必要なテストプレイの数も増えてしまいますが、長く遊べるゲームを作るには有効な方法です。

ということで、「自機は地味だけれども、ゲームの魅力の半分くらいは自機で決まる！」というのが本章のまとめです。

Stage 04 ▶▶

武器 Weapon

世の中には、ひたすら弾をよけるだけのストイックなゲームもありますが、ほとんどのゲームではなんらかの攻撃手段があり、敵を倒すことを目的としています。本章ではこういった攻撃手段、すなわち自機が持つ「武器」について解説します。

シューティングゲームの大きな魅力は、ルールがシンプルで奥が深いところです。なかには複雑なゲームもありますが、多くのゲームでは「通常武器(ショット)+特殊武器(ボム)」という構成を基本としています。遊ぶ前に念入りにマニュアルを読まなくても、「とりあえず左側のボタンがショットで、右側はきっとボムでしょ?」と見当がつくのがシューティングゲームのよいところだと思います。筆者が極端に無精なのかもしれませんが、ゲームセンターなどではインスト(説明書)をゆっくり読めないことも多いでしょうから、ルールを知らなくても遊べるのは重要です。

本章では自機が持つ武器のうち、ショットについて解説します。オーソドックスなショットのほかに、ロックオンレーザー(誘導レーザー)やロックオンショットといった特殊武器、あるいは溜め撃ちやコマンド撃ちといった特殊な撃ち方についても触れます。なお、ボムについては「特殊攻撃」の章で説明します。

基本のショット操作

多くのゲームにおいて、自機の基本的な武器はショットです。ショットボタンを押すたびに、自機の前方や斜め前方に向かって弾が発射されます(Fig. 4-1、4-2)。ここでは自動連射や溜め撃ちといった機能はさておき、まずは1回ボタンを押すたびに1回だけ弾が出るような、昔ながらのシンプルなショットについて考えます。

Fig. 4-1 前方に1発だけ出るショット

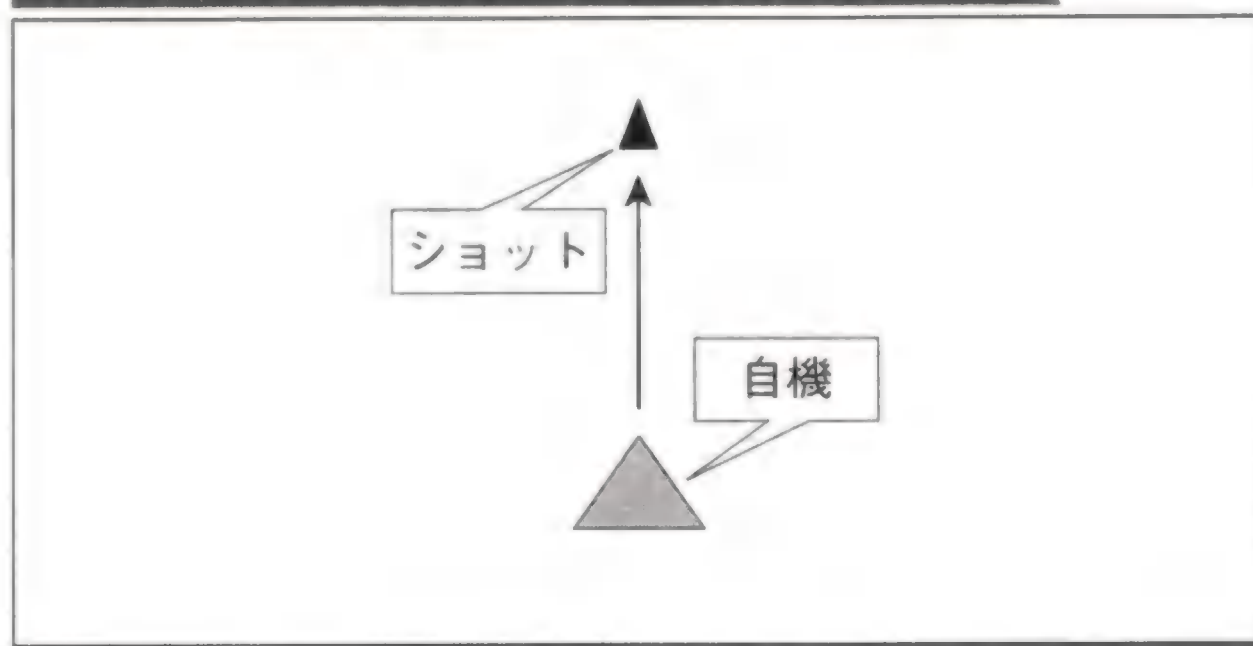
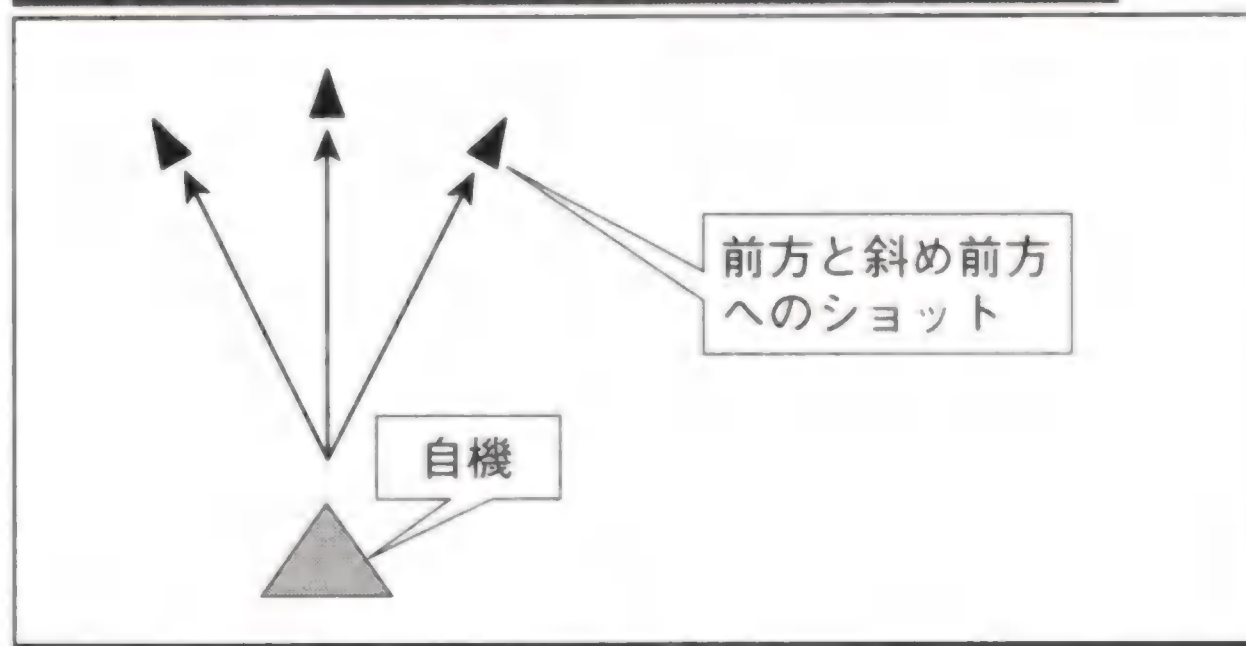



Fig. 4-2 前方と斜め前方に出るショット

「ボタンを押したらショットが出る」という動作は、単純に考えると次のような方法で実現できそうです。

①ボタンが押されているかどうかを調べる
②ボタンが押されていたら弾を1回撃つ

ところが、この方法には問題があります。この方法では、1回ボタンを押しただけなのに、弾が複数回出てしまうことがあります。その理由はFig. 4-3のとおりです。シューティングゲームでは、1/60秒などの短い時間間隔でスティックやボタンの状態を調べます。そのため、人間が1回だけボタンを押している間に、2回以上「ボタンが押されている」という状態が検出されることがあります。この場合、「ボタンが押されていたら弾を1回撃つ」という方法では、2回以上弾を撃ってしまうことになるのです。

この方法をFig. 4-4のように改良すると、1回ボタンを押すたびに1回だけ弾が出るようにできます。ポイントは「前回ボタンが押されておらず、今回ボタンを押したときだけ弾を発射する」ということです。これなら、ボタンを短く押したときはもちろん、ボタンを押しっぱなしにしたときでも、ボタンを押した最初の瞬間に1回だけ弾が出るようになります。

Fig. 4-4の方法をプログラムにすると、List 4-1のようになります。前回のボタンの状態を保存しておくことがポイントです。

サンプル

基本のショット操作　→ P. 316

Fig. 4-3 1回ボタンを押しただけなのに複数発の弾が出てしまう理由

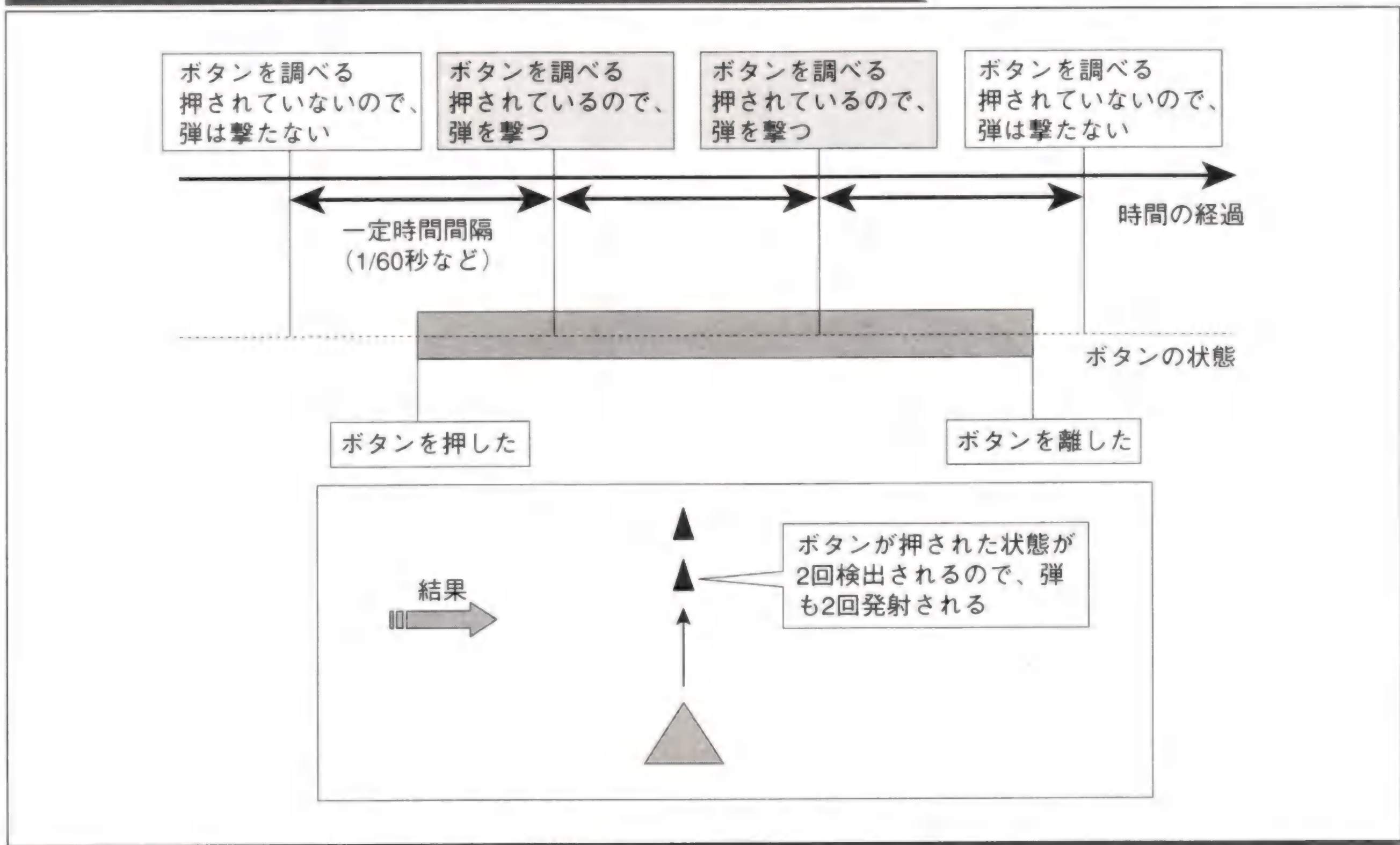
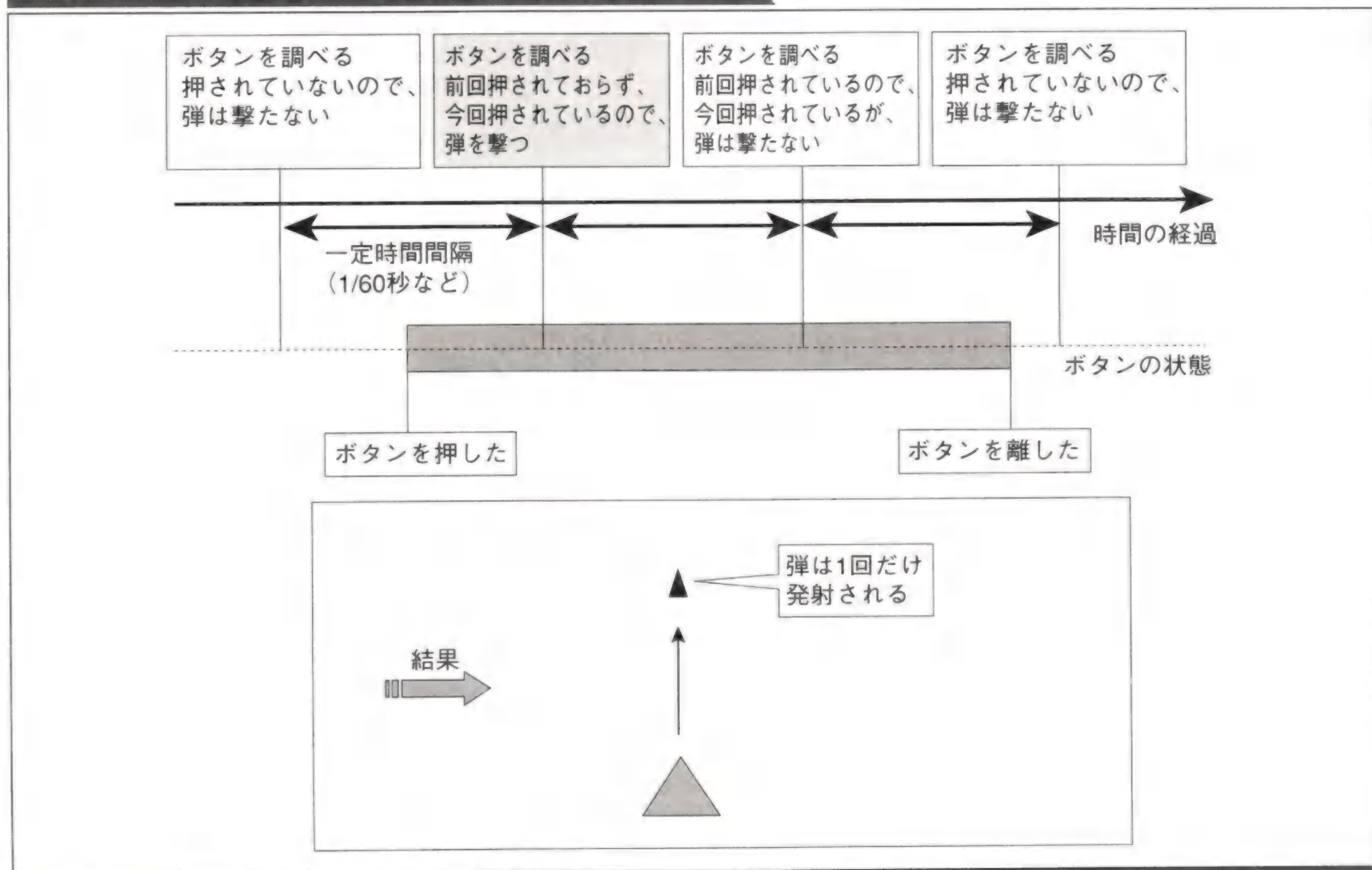



Fig. 4-4 1回のボタンで1回だけ弾が出るようにする方法

List 4-1 基本のショット操作

```
void BasicShot(
    bool button   // 今回のボタンの状態（押されたときtrue）
) {
    // 前回のボタンの状態（押されたときtrue）
    static bool prev_button=false;

    // ショットの発射：
    // 前回ボタンが押されておらず、今回押されているときだけ、
    // ショットを発射する。
    // 発射の詳細な処理はShot関数で行うものとする。
    if (!prev_button && button) Shot();

    // 今回のボタンの状態を保存する
    prev_button=button;
}
```

連射

昔のゲームでは、ボタンを「こする」ようにして連射するものが少なくなく、こういった「連射の能力」もゲーマーとして必要な素養の1つでした。しかし、今のゲームにはなんらかの自動連射機能がついているものが多く、手動で連射する必要はほとんどなくなりました。

連射の仕組みは簡単です。実は、「基本のショット」(→ P. 114) で解説した1番目の方法 (Fig. 4-3) がそのまま連射機能になります。この方法では、ボタンを押しているかぎりショットが自動的に連射されます。Fig. 4-3の方法は、1回ボタンを押したときに1回だけショットを撃つ、という処理には不向きですが、実は連射機能そのものなのです。

ということで、Fig. 4-3をプログラムにしたList 4-2が、連射機能のプログラムということになります。この処理はList 4-1のような「1回押しで1回発射」の処理とは違うので、単発と連射の両方のショットを使いたいときには、ボタンを2個使ってそれぞれに別々のショットを割り当てます。

サンプル

連射 → P. 316

List 4-2 連射

```
void AutoShot(
    bool button  // ボタンの状態（押されたときtrue）
) {
    // ショットの発射：
    // ボタンが押されていたらショットを発射する。
    // 発射の詳細な処理はShot関数で行うものとする。
    if (button) Shot();
}
```

溜め撃ち

ボタンを押しっぱなしにしてパワーを溜め、通常よりも強力なショットを撃つのがいわゆる「溜め撃ち」です。溜め撃ちがあるかどうかはゲームによりますが、最近では、ボタンを押しっぱなしにするとなんらかの特殊なショットが出るようになっているゲームが多くあります。

基本的な溜め撃ちの手順は次のとおりです。ここでは便宜上、溜めの度合いのことを「溜めパワー」と呼ぶことにします。

①ボタンを押すと溜めが始まる
②時間経過とともに溜めパワー（溜めの度合い）が増える
③ボタンを離すと、溜めパワーの値に応じた強さや大きさの弾が発射される

この手順を詳しくしたものがFig. 4-5です。ボタンを押している間は溜めパワーを増やします。ボタンを離したら、その時点における溜めパワーの値に応じて、撃つ弾の強さや大きさを決めます。

溜めパワーが少なくても小さな弾が出るゲームもあれば、溜めパワーが一定値以上でないとまったく弾が出ないゲームもあります。また、溜めパワーは無制限に増えていくわけではなく、たいていの場合は上限が設けられています。上限まで溜めパワーを増やした場合には、通常の溜め撃ち弾とは違った特別な弾が出るゲームもあります（「フェリオス（→ P. 333）」など）。

List 4-3は溜め撃ちの処理をまとめたプログラムです。

サンプル

溜め撃ち　→ P. 316

Fig. 4-5 溜め撃ちの仕組み

ボタンを調べる
押されておらず、
溜めパワーが0なので、
弾は撃たない
（溜めパワー→0）

ボタンを調べる
押されているので、
溜めパワーを増やす
（溜めパワー→1）

ボタンを調べる
押されているので、
溜めパワーを増やす
（溜めパワー→2）

ボタンを調べる
押されているので、
溜めパワーを増やす
（溜めパワー→3）

ボタンを調べる
押されておらず、
溜めパワー>0なので、
溜めパワーを0にして、
溜め撃ち弾を撃つ
（溜めパワー→0）

時間の経過

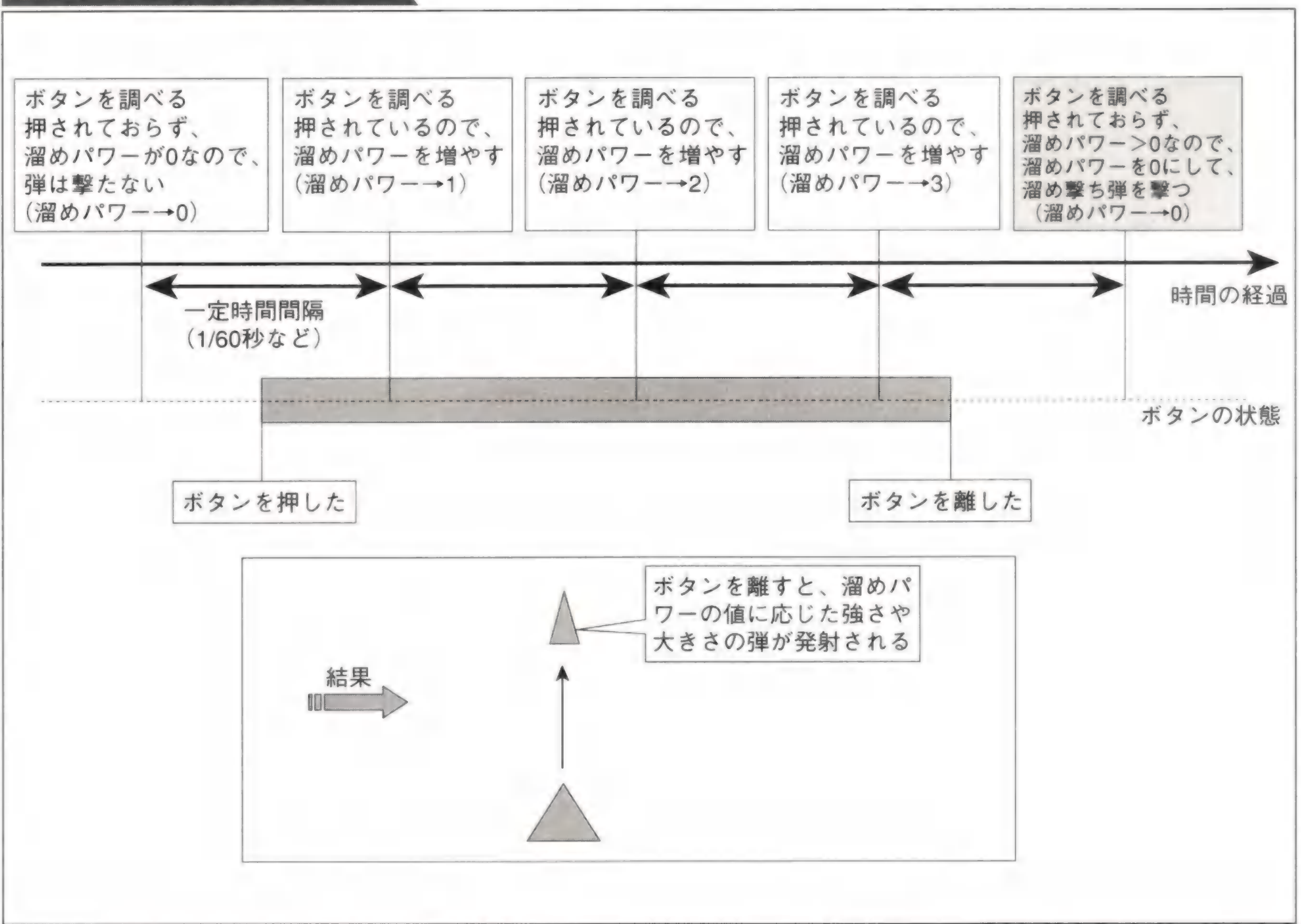
一定時間間隔
（1/60秒など）

ボタンの状態

ボタンを押した

ボタンを離した

結果

ボタンを離すと、溜めパワーの値に応じた強さや大きさの弾が発射される

List 4-3 溜め撃ち

```
void PowerShot(
    bool button,     // ボタンの状態（押されたときtrue）
    int min_power,   // 弾が出る最小のパワー
    int max_power    // 最大のパワー
) {
    // 溜めパワー
    static int power=0;

    // ボタンを押している場合：
    // 溜めパワーを増やす。
    if (button && power<max_power) power++;

    // ボタンを離している場合：
    // 溜めパワー>0ならば弾を撃ち、溜めパワーを0に戻す。
    if (!button && power>0) {

        // 最大パワーのとき：
```

```
        // 特別な弾を撃つ。
        // 具体的な処理はMaxShot関数で行うとする。
        if (power==max_power) MaxShot(); else

        // 最小パワー以上、最大パワー未満のとき：
        // 溜めパワーに応じた強さの溜め撃ち弾を撃つ。
        // 具体的な処理はBigShot関数で行うとする。
        if (min_power<=power) BigShot(power);

        // 溜めパワーを0に戻す
        power=0;

    }
}
```

連射と溜め撃ちの共存(セミオート連射)

連射と溜め撃ちは、普通は同じボタンに割り当てることはできません。これは、どちらもボタンを押しっぱなしにする必要があるからです。したがって、連射と溜め撃ちの両方があるゲームでは、連射と溜め撃ちをそれぞれ別のボタンに割り当てなければなりませんが、これは使うボタンの数が増えて好ましくありません。シューティングゲームは「1スティック+2ボタン」が基本なので、かぎられた数のボタンを大事に使いたいのです。

そこで、連射と溜め撃ちを同じボタンに割り当てることを考えます。実際に最近のシューティングゲームでは、連射と溜め撃ち（あるいはレーザーなど）を同じボタンに割り当てているものが少なくありません。こういったゲームの多くは次のように操作します。

- **ボタンをある程度の速度で連打すると連射になる**
- **ボタンを押しっぱなしにすると溜め撃ちになる**

「ある程度の速度で連打すると連射になる」というのがポイントです。1回ボタンを押すと数発の弾が出るので、それが途切れる前に一度ボタンを離してまた押すと、間を空けることなく弾を連射することができます。昔のゲームとは違って、ボタンをこするように連射する必要はなく、リズミカルに「タンタンタン……」とボタンを叩いていればよいのです。このような連射方法は「セミオート連射」と呼ばれることがあります。

■ 連射処理

連射と溜め撃ちを共存させる方法は少し複雑です。まず、ボタンをリズミカルに叩いたときに連射（セミオート連射）を行う方法を考えます（Fig. 4-6）。溜め撃ちのときには溜めパワーと

いう概念を導入しましたが（→ P. 117）、今度は溜めパワーに加えて「連射ゲージ」という概念も導入します。連射ゲージは、ボタンを離してから次にボタンを押すまでに連射を継続させる時間を表します。

セミオート連射では、ボタンを押したら連射を開始します（Fig. 4-6-①）。ボタンを押している間は連射を継続し（Fig. 4-6-②）、ボタンを離しても連射ゲージが0より大きい場合には連射を続けます（Fig. 4-6-③）。連射ゲージが0になる前に再びボタンを押せば、連射はずっと続きます（Fig. 4-6-④）。なお、Fig. 4-6では連射ゲージの最大値を3としました。

次のFig. 4-7は連射が止まる仕組みです。ボタンを離していると、連射ゲージは減っていきます（Fig. 4-7-①②③）。連射ゲージが0より大きいかぎりは連射が続きますが、連射ゲージが0になると連射は止まります（Fig. 4-7-④）。連射を続けるには、連射ゲージが切れる前に再びボタンを押す必要があります。

なお、Fig. 4-6や4-7では連射ゲージの最大値を3としましたが、実際にはもう少し大きめの値を使うことになるでしょう。たとえば1/60秒単位で動くゲームで、連射を継続するために1秒につきボタンを4回叩くことにするならば、連射ゲージの最大値を15（=60/4）に設定します。

Fig. 4-6 連射（セミオート連射）

ボタンを調べる
押されておらず、
溜めパワーが不足、
連射ゲージも0なので、
溜めパワーを0にして、
弾は撃たない
（溜め→0、連射→0）

①
ボタンを調べる
押されており、
溜めパワー不足なので、
溜めパワーを増やし、
連射ゲージを3にして、
弾を撃つ
（溜め→1、連射→3）

②
ボタンを調べる
押されており、
溜めパワー不足なので、
溜めパワーを増やし、
連射ゲージを3にして、
弾を撃つ
（溜め→2、連射→3）

③
ボタンを調べる
押されておらず、
溜めパワー不足、
連射ゲージ>0なので、
溜めを0とし、連射を−1
して、弾を撃つ
（溜め→0、連射→2）

④
ボタンを調べる
押されており、
溜めパワー不足なので、
溜めパワーを増やし、
連射ゲージを3にして、
弾を撃つ
（溜め→1、連射→3）

一定時間間隔
（1/60秒など）
時間の経過
ボタンの状態
ボタンを押した
ボタンを離した
ボタンを押した
結果
リズミカルにボタンを叩くと、
途切れなく連射できる

連射ゲージの最大値を大きくするほどボタンを叩く速度が遅くてすみますが、ボタンを離してから連射が止まるまでの反応が鈍くなります。セミオート連射があるゲームを遊ぶときによく観察すると、ボタンを離してからショットが止まるまでに少し時間がかかることに気づくでしょう。

Fig. 4-7 連射の停止

ボタンを調べる
押されており、
溜めパワー不足なので、
溜めパワーを増やし、
連射ゲージを3にして、
弾を撃つ
(溜め→1、連射→3)

①
ボタンを調べる
押されておらず、
溜めパワー不足、
連射ゲージ>0なので、
溜めを0とし連射を1
として、弾を撃つ
(溜め→0、連射→1)

②
ボタンを調べる
押されておらず、
溜めパワー不足、
連射ゲージ>0なので、
溜めを0とし連射を1
として、弾を撃つ
(溜め→0、連射→1)

③
ボタンを調べる
押されておらず、
溜めパワー不足、
連射ゲージ>0なので、
溜めを0とし連射を1
として、弾を撃つ
(溜め→0、連射→0)

④
ボタンを調べる
押されておらず、
溜めパワー不足、
連射ゲージも0なので、
溜めパワーを0にして、
弾は撃たない
(溜め→0、連射→0)

時間の経過
一定時間間隔
(1/60秒など)
ボタンの状態
ボタンを離した
結果
ボタンを離すと、
少ししてから連射が止まる

■ 溜め撃ち処理

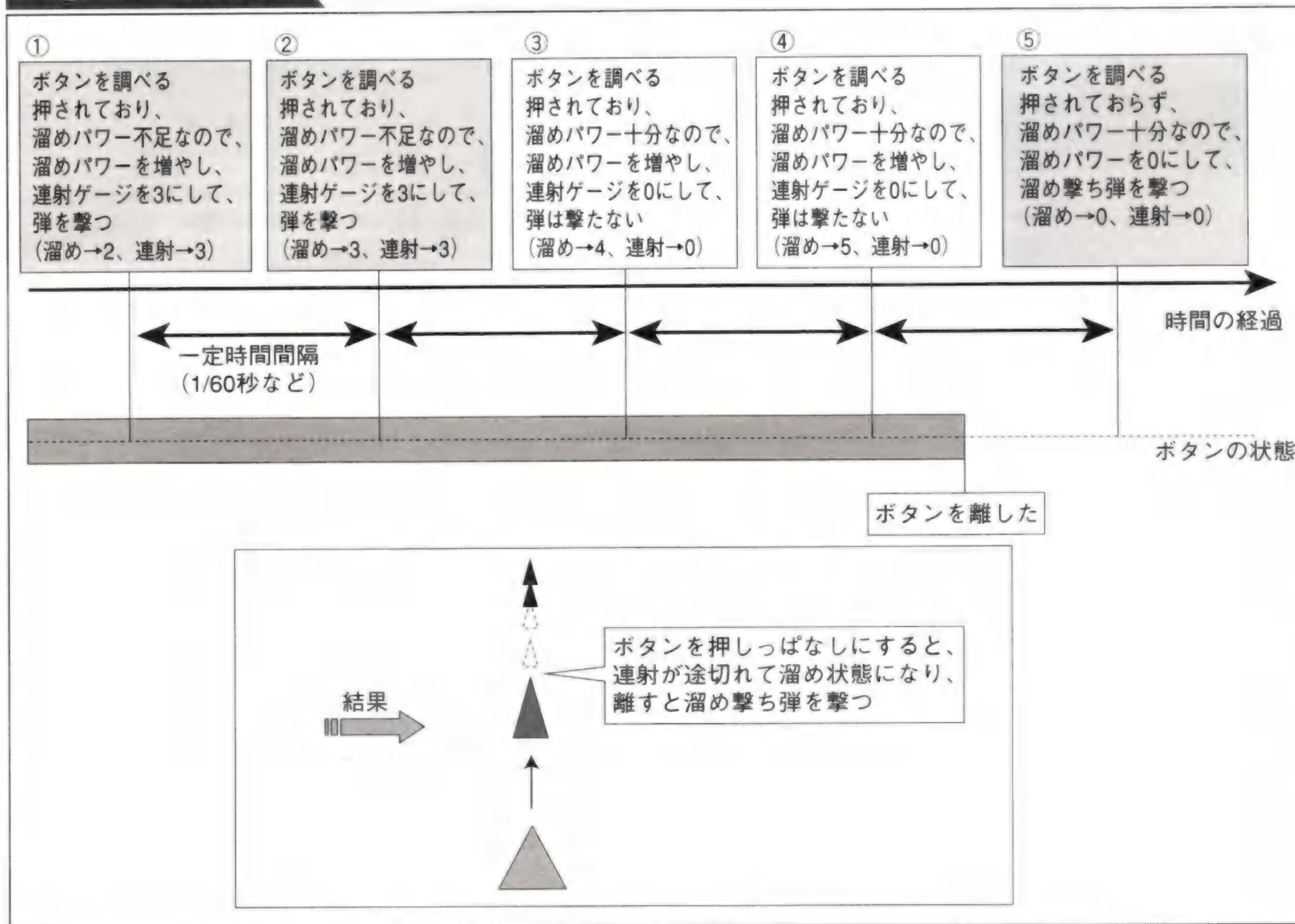
次のFig. 4-8が溜め撃ちの仕組みです。ボタンを押している間は、溜めパワーが増えていきます。しかし、溜めパワーが一定値未満の場合には、溜め状態にはならず、弾は連射されます(Fig. 4-8-①②)。溜めパワーが一定値以上になると、連射が止まって溜め状態になります(Fig. 4-8-③④)。溜め状態に入ったあとにボタンを離すと、溜めパワーの値に応じた強さの溜め撃ち弾が放たれます(Fig. 4-8-⑤)。

Fig. 4-8では溜め状態に入るために必要な最小パワーを3としましたが、これは1/60秒周期のゲームでは約1/20秒の溜め時間(3=60/20)に相当します。溜め時間があまり短いと、撃つ弾がすぐに溜め撃ち弾になってしまい、普通の弾を連射するのが難しくなるので、実際のゲームでは最小パワーをもう少し大きめにします。また、溜めパワーの最小値と連射ゲージの最大値とは同じ値か近い値にするのがよいでしょう。

だいぶ複雑でしたが、以上が連射（セミオート連射）と溜め撃ちを共存させる仕組みです。最後に、この仕組みをプログラムにまとめます（List 4-4）。

サンプル

セミオート連射 → P. 316

Fig. 4-8 溜め撃ち

List 4-4 連射と溜め撃ちの共存

```
void SemiAutoShot(
    bool button,     // ボタンの状態（押されたときtrue）
    int min_power,   // 溜めパワーの最小値
    int max_power,   // 溜めパワーの最大値
    int max_gauge    // 連射ゲージの最大値
) {
    // 溜めパワー、連射ゲージ
    static int power=0, gauge=0;

    // ボタンを押している場合
```

```
    if (button) {

        // 溜めパワー不足の場合：
        // 通常の弾を撃つ。
        // 具体的な処理はShot関数で行うとする。
        if (power<min_power) Shot();

        // 溜めパワー十分の場合：
        // 溜め状態の表示をする。
        // 具体的な処理はPowerEffect関数で行うとする。
        else PowerEffect();

        // 溜めパワーを増やし、連射ゲージを最大にする
        if (power<max_power) power++;
        gauge=max_gauge;
    }

    // ボタンを離している場合：
    else {

        // 溜めパワー十分の場合：
        // 溜め撃ち弾を撃つ：
        // 最大パワーのときは特別な弾を撃つ。
        // 具体的な処理はMaxShot関数とBigShot関数で行うとする。
        if (min_power<=power) {
            if (power==max_power) MaxShot();
                else BigShot(power);
            gauge=0;
        }

        // 連射ゲージ>0の場合：
        // 通常の弾を撃ち、連射ゲージを-1する。
        // 具体的な処理はShot関数で行うとする。
        if (gauge>0) {
            Shot();
            gauge--;
        }

        // 溜めパワーを0にする
        power=0;
    }
}
```

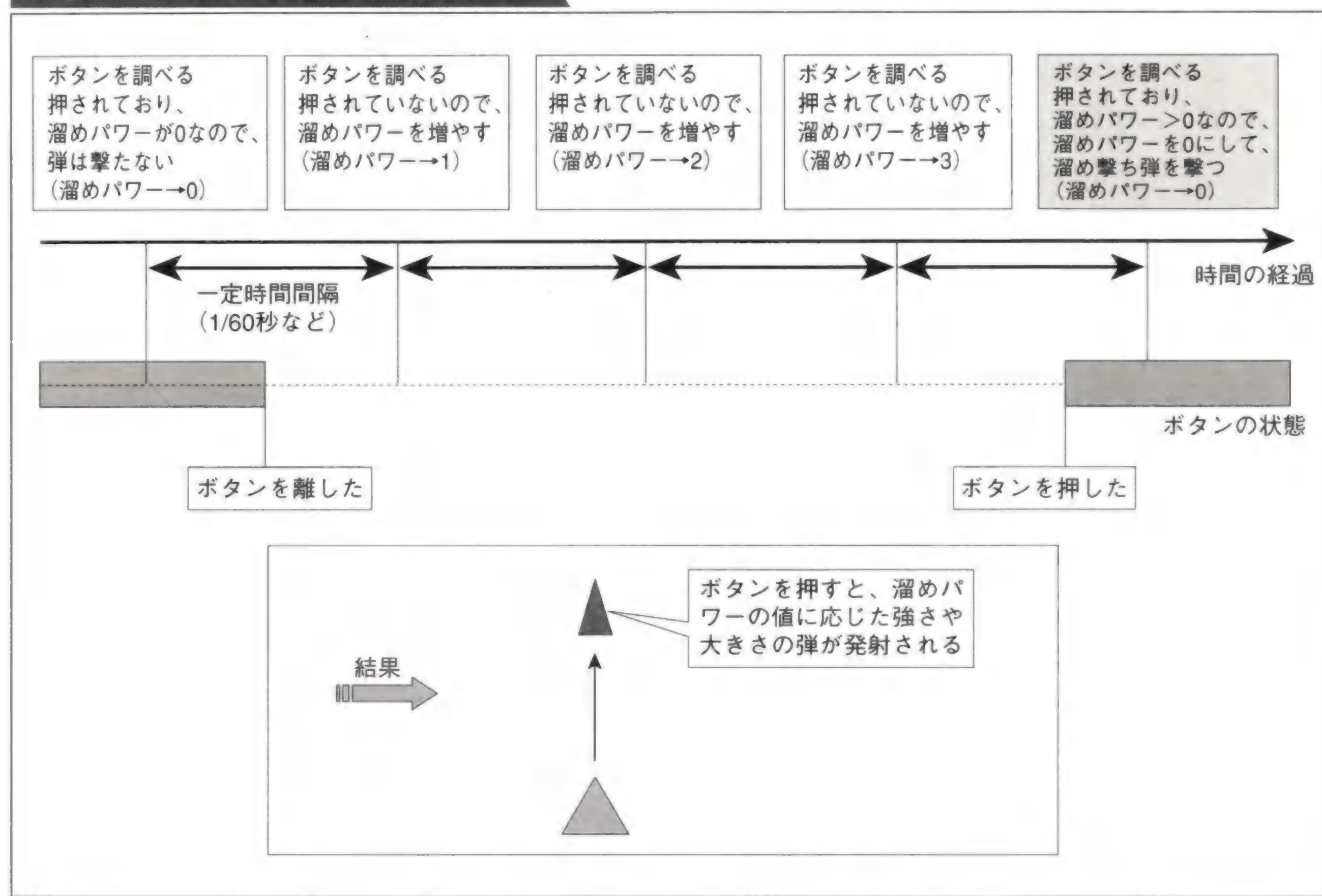
ボタンを離して溜める溜め撃ち

ボタンを押してパワーを溜める溜め撃ち(→ P. 117)とは逆に、ボタンを離して溜める溜め撃ちもあります(Fig. 4-9)。「ドラゴンセイバー(→ P. 332)」などはこの方式です。ボタンを離している間はパワーが溜まっていき、パワーが十分溜まったあとにボタンを押すと、溜め撃ち弾が発射されます。

Fig. 4-9の手順をプログラムにしたものがList 4-5です。

サンプル

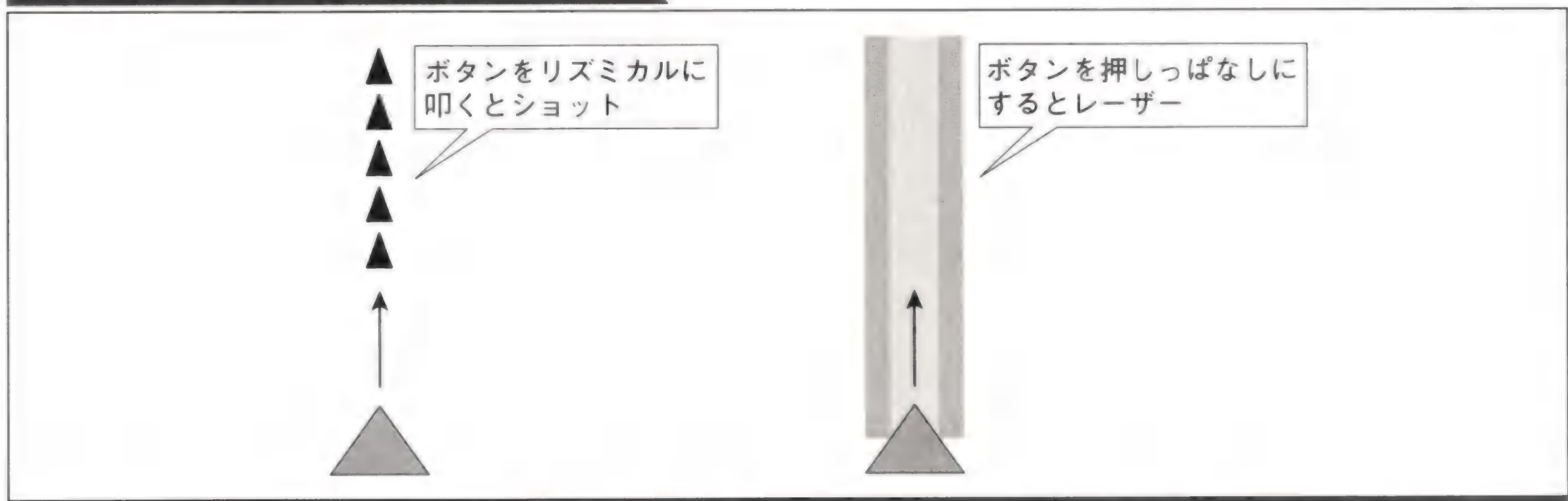
ボタンを離して溜める溜め撃ち　→ P. 317

Fig. 4-9　ボタンを離して溜める溜め撃ち

List 4-5 ボタンを離して溜める溜め撃ち

```
void PowerShotReleased(
    bool button,    // ボタンの状態（押されたときtrue）
    int min_power,  // 弾が出る最小のパワー
    int max_power   // 最大のパワー
) {
    // 溜めパワー
    static int power=0;

    // ボタンを離している場合：
    // 溜めパワーを増やす。
    if (!button && power<max_power) power++;

    // ボタンを押している場合：
    // 溜めパワー>0ならば弾を撃ち、溜めパワーを0に戻す。
    if (button && power>0) {

        // 最大パワーのとき：
        // 特別な弾を撃つ。
        // 具体的な処理はMaxShot関数で行うとする。
        if (power==max_power) MaxShot(); else

        // 最小パワー以上、最大パワー未満のとき：
        // 溜めパワーに応じた強さの溜め撃ち弾を撃つ。
        // 具体的な処理はBigShot関数で行うとする。
        if (min_power<=power) BigShot(power);

        // 溜めパワーを0に戻す
        power=0;
    }
}
```

連射とレーザーの共存

連射と溜め撃ちを使い分けるゲームのほかに、連射とレーザーを撃ち分けるゲームもあります。これは、ボタンをリズミカルに叩くと通常弾の連射になり、押しっぱなしにすると威力が高いレーザーが出るというものです（Fig. 4-10）。

Fig. 4-10 連射とレーザーの組み合わせ

このように、連射とレーザーを組み合わせたゲームの代表格は「首領蜂（→ P. 331）」シリーズ（「怒首領蜂」「怒首領蜂大往生」など）です。「首領蜂」シリーズのゲームデザインで秀逸なのは、「レーザー発射中には自機の移動速度が遅くなる」ということです。自機の速度が遅くなると画面内を大きく移動するのは難しくなりますが、細かい弾よけをするのには向いています。

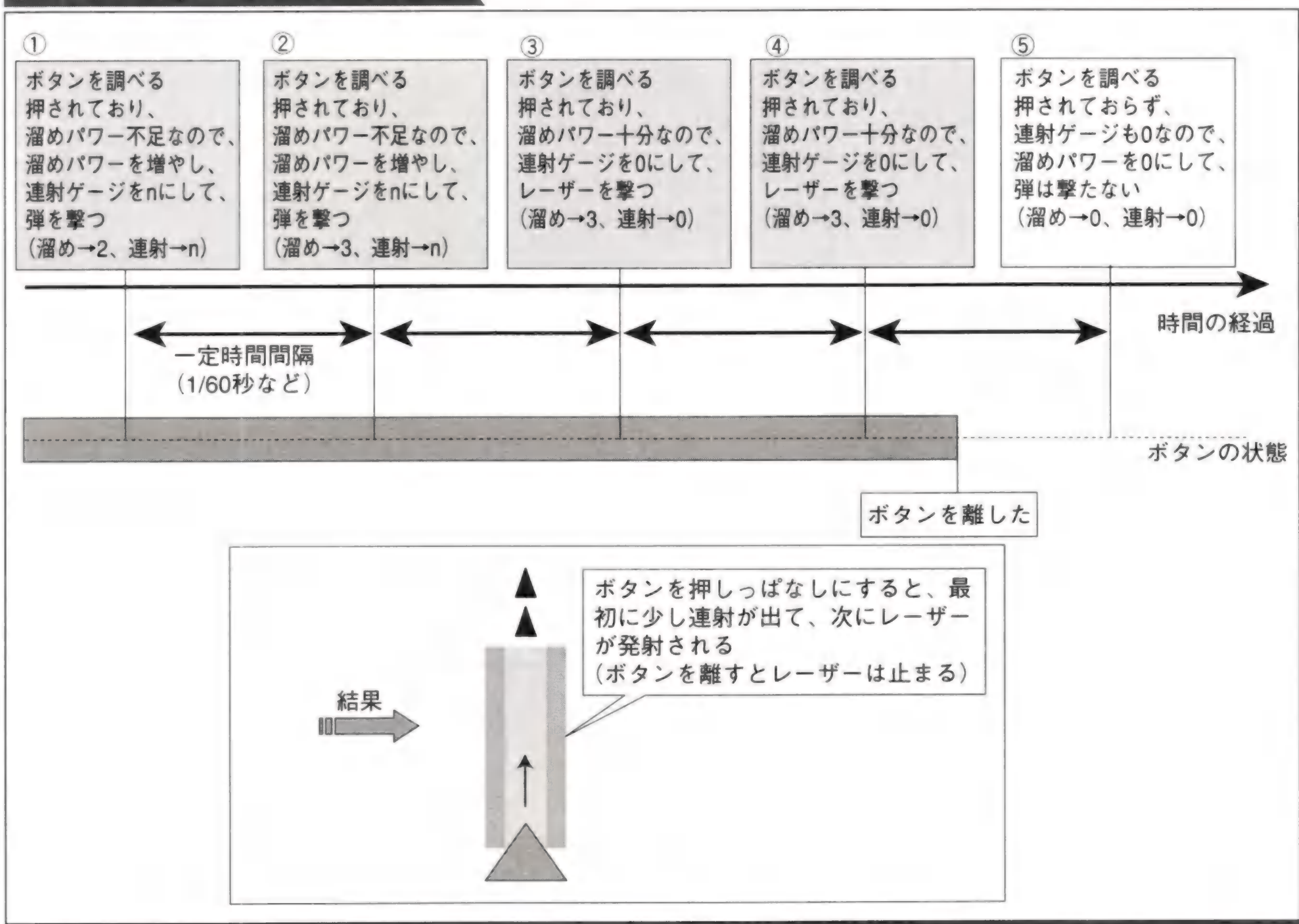
逆に通常弾を撃っている間は、威力は弱いのですが、自機の移動速度が速くなり、また広範囲に撃つことができるので、弱い敵を効率よく片づけることができます。そのため、

- **小型機の掃討や大きな回避には通常弾の連射**
- **中型～大型機の破壊やボスキャラの削り、細かい弾よけにはレーザー**

のように連射とレーザーを切り替えながら戦う必要があり、これが絶妙なゲームバランスを生み出しているのです。

連射とレーザーとを1つのボタン上で共存させる仕組みは、連射と溜め撃ちを1つのボタン上で共存させる方法（→ P. 119）に似ています（Fig. 4-11）。

Fig. 4-11はボタンを押しっぱなしにしたときの状況を示しています。連射と溜め撃ちを共存させるときと同様に、ここでは溜め撃ちの溜め度合いを表す「溜めパワー」と、連射の残存時間を表す「連射ゲージ」という概念を使います。

Fig. 4-11 連射とレーザーの共存

ボタンを押している間は、溜めパワーが増えていきます(Fig. 4-11-①②)。溜めパワーがレーザー発射の必要値に達する以前には、レーザーは発射されず、かわりに通常弾が連射されます。

溜めパワーが必要値に達すると、通常弾の連射は止まり、かわりにレーザーが発射されます(Fig. 4-11-③④)。ここでは溜めパワーの必要値を3としました。ボタンを離すとレーザーは止まります(Fig. 4-11-⑤)。

連射とレーザーを共存させる仕組みをまとめたプログラムはList 4-6です。連射と溜め撃ちの共存に関するプログラム(List 4-4 → P. 122)と見比べると、似ている部分が多いことに気づくでしょう。

サンプル

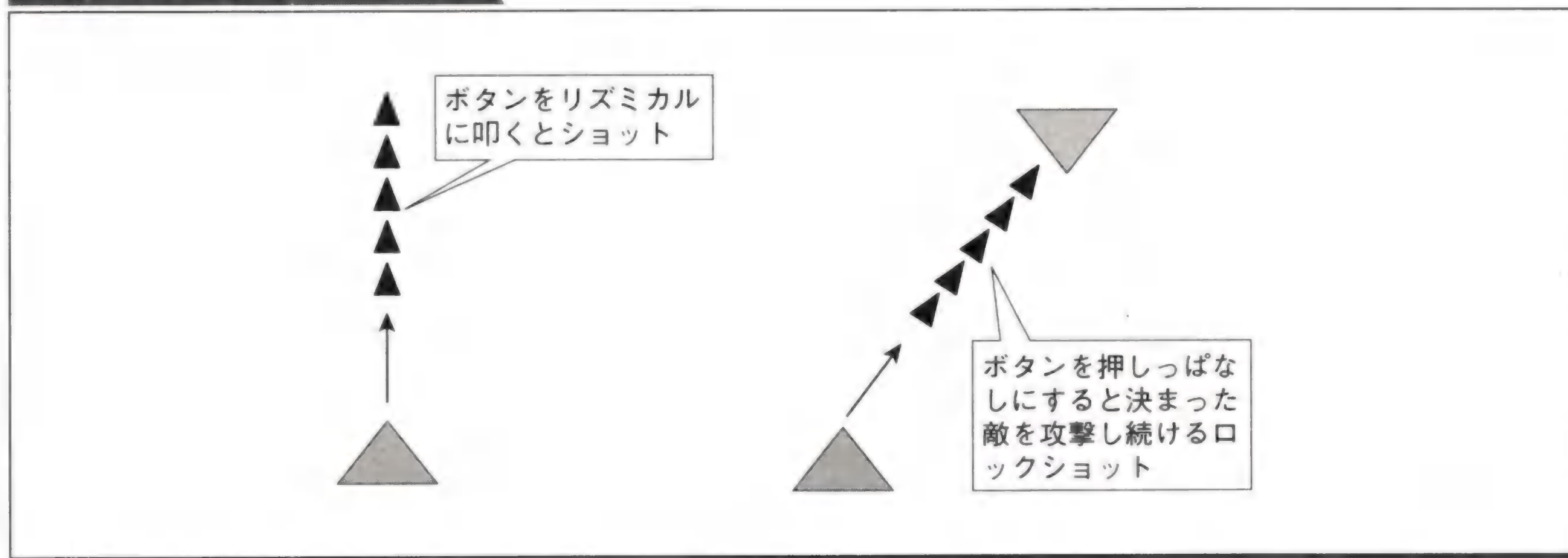
● 連射とレーザーの共存 → P. 317

List 4-6 連射とレーザーの共存

```
void ShotAndLaser(
    bool button,     // ボタンの状態（押されたときtrue）
    int min_power,  // 溜めパワーの最小値
    int max_gauge   // 連射ゲージの最大値
) {
    // 溜めパワー、連射ゲージ
    static int power=0, gauge=0;

    // ボタンを押している場合
    if (button) {

        // 溜めパワー不足の場合：
        // 溜めパワーを増やし、連射ゲージを最大にして、
        // 弾を撃つ。具体的な処理はShot関数で行うとする。
        if (power<min_power) {
            power++;
            gauge=max_gauge;
            Shot();
        }

        // 溜めパワー十分の場合：
        // 連射ゲージを0にして、レーザーを撃つ。
        // 具体的な処理はLaser関数で行うとする。
        else {
            gauge=0;
            Laser();
        }
    }

    // ボタンを離している場合：
    else {

        // 連射ゲージ>0の場合：
        // 普通の弾を撃ち、連射ゲージを-1する。
        if (gauge>0) {
            Shot();
            gauge--;
        }

        // 溜めパワーを0にする
        power=0;
    }
}
```

ロックショット

ロックショットはレーザーに似ていますが、こちらは一度当てた敵を自動的に攻撃し続けるショットです。連射とレーザーを撃ち分ける場合（→ P. 126）と同様に、ボタンをリズミカルに叩くと通常弾の連射になり、押しっぱなしにするとロックショットが出る、という操作になっています（Fig. 4-12）。

Fig. 4-12 ロックショット

ロックショットは「ケツイ（→ P. 327）」や「プロギアの嵐（→ P. 333）」などに入っている要素です。「ケツイ」の場合は自機に2機のオプションがついていて、そのオプションからロックショットが出るようになっています。ロックショットの使い方はレーザーに似ていて、主に中型機や大型機の破壊に使います。「ケツイ」ではさらに、敵を破壊したのが通常ショットなのかロックショットなのかに応じて、得点の計算方法が変わるようになっています。

ロックショットはレーザーと同じ方法で発射することができます。基本的には、List 4-6のレーザーを撃つ処理（Laser関数）を、ロックショットを撃つ処理に変更するだけです。

一方、発射されたあとのロックショットの動きは少し複雑です。ロックショットは敵に当たった時点から、その敵の追尾を開始します。以後は敵や自機が動いても、自動的に敵を追尾するようになります（Fig. 4-13）。

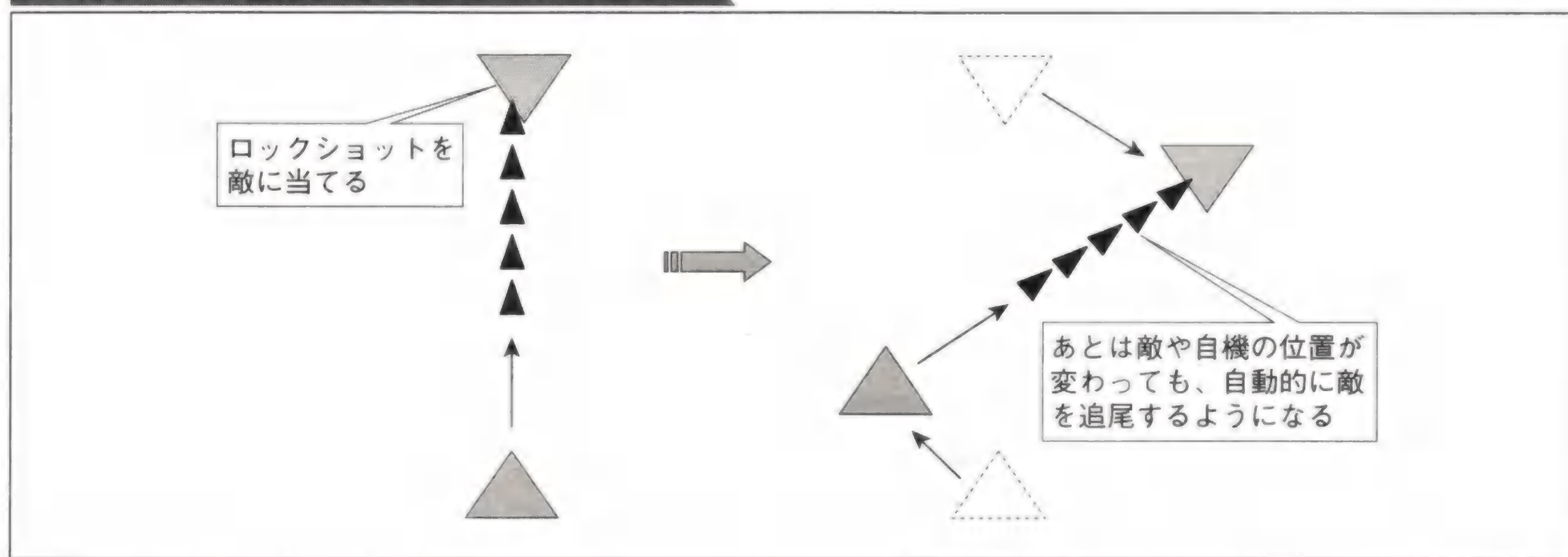
List 4-7はロックショットの仕組みをまとめたプログラムです。基本的にはList 4-6と同じなのですが、ロックショットが敵を追尾しているかどうかによって、ロックショットを撃つ方向が違います。追尾しているときには敵に向かってショットを撃ち、追尾していないときには正面にショットを撃ちます。敵に向かって撃つショットは「狙い撃ち弾」（→ P. 10）、正面に撃つショットは「方向弾」（→ P. 23）と同じ方法で実現できます。

サンプル

● ロックショット → P. 317

Fig. 4-13 発射されたロックショットの動き

ロックショットを
敵に当てる

あとは敵や自機の位置が
変わっても、自動的に敵
を追尾するようになる

List 4-7 ロックショット

```
void LockShot(
    bool button,      // ボタンの状態（押されたときtrue)
    int min_power,    // 溜めパワーの最小値
    int max_gauge,    // 連射ゲージの最大値
    int enemy_id,     // ロックショットが追尾している敵のID
                      // （追尾していないときには負の値）
    float front_dir   // 自機の正面に相当する方向
) {
    // 溜めパワー、連射ゲージ
    static int power=0, gauge=0;

    // ボタンを押している場合
    if (button) {

        // 溜めパワー不足の場合：
        // 溜めパワーを増やし、連射ゲージを最大にして、
        // 弾を撃つ。具体的な処理はShot関数で行うとする。
        if (power<min_power) {
            power++;
            gauge=max_gauge;
            Shot();
        }

        // 溜めパワー十分の場合：
        // 連射ゲージを0にして、ロックショットを撃つ。
        else {
            gauge=0;

            // ロックショットを撃つ：
            // 敵を追尾しているときは敵を狙い撃ちする。
```

```
            // （「自機に向かって飛ぶ弾」と同様）
            // 敵を追尾していないときは自機の正面に弾を撃つ。
            // （「自由な方向に飛ぶ弾」と同様）
            // それぞれ具体的な処理はAimingShot関数と
            // DirectedShot関数で行うとする。
            if (enemy_id>=0) {
                AimingShot(enemy_id);
            } else {
                DirectedShot(front_dir);
            }
        }
    }

    // ボタンを離している場合：
    else {

        // 連射ゲージ>0の場合：
        // 普通の弾を撃ち、連射ゲージを-1する。
        if (gauge>0) {
            Shot();
            gauge--;
        }

        // 溜めパワーを0にする
        power=0;
    }
}
```

コマンドショット

一部のシューティングゲームでは、格闘ゲームのようなコマンド入力によって特殊なショットを出すことができるようになっています。とはいえ、本物の格闘ゲームほど複雑なコマンドを入力させるゲームはなく、せいぜい、いわゆる「波動拳コマンド」や「昇龍拳コマンド」程度のものです（Fig. 4-14）。コマンドショットを導入すると、シューティングゲームを普通とはちょっと違った味つけにできます。ただし、コマンドショットはプレイヤーの好き嫌いが分かれそうな要素なので、濫用するのは危険です。

ちなみに、コマンドとしては少し単純ですが、「サイヴァリア（→ P. 327）」シリーズには「ローリング」という特殊操作があります。これはレバーを素早く左右に入れることによって自機が回転し、ショットが一時的に強くなるというものです。これも一種のコマンドショットといえるでしょう。

Fig. 4-14 コマンドショットの例

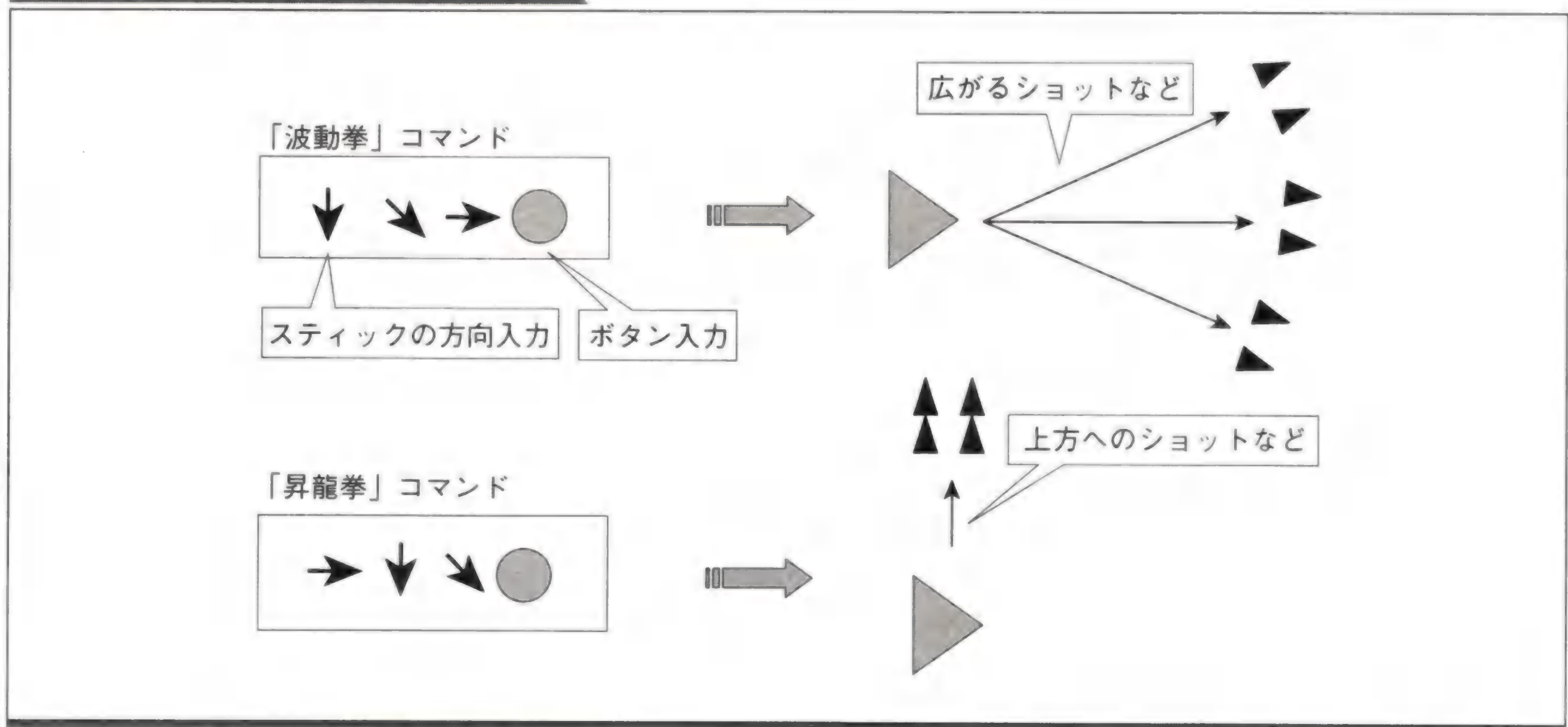
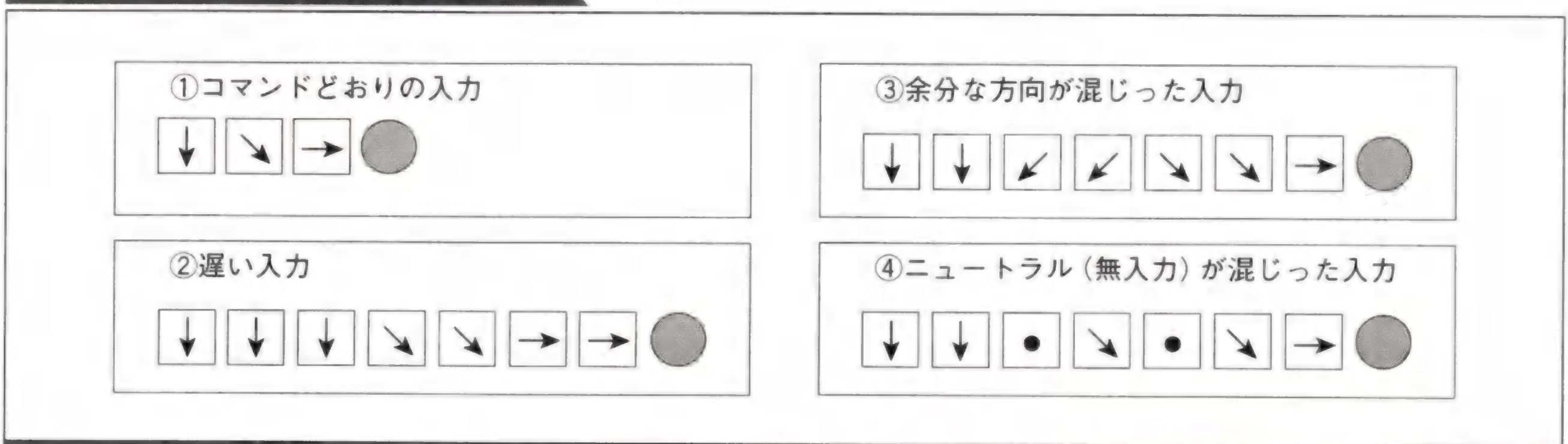



Fig. 4-15 コマンド入力のゆらぎ

■ コマンド入力のゆらぎ

コマンドショットを扱うときには、「コマンドは常に正確に入力されるわけではない」ということに注意が必要です（Fig. 4-15）。基本となるコマンド入力（Fig. 4-15-①）は1通りでも、実際にプレイヤーがスティックを使って入力するパターンがそのとおりになることは、滅多にありません。

シューティングゲームでは、非常に短い時間間隔（たとえば1/60秒周期）でスティックの入力を読み取ります。そのため、たとえばFig. 4-15-①のとおりにコマンド入力しようとすると、1方向1ボタンあたり正確に1/60秒ずつ、全部でわずか1/15秒の間にコマンド入力を完了しなければなりません。これをプレイヤーに求めるのは酷でしょう。

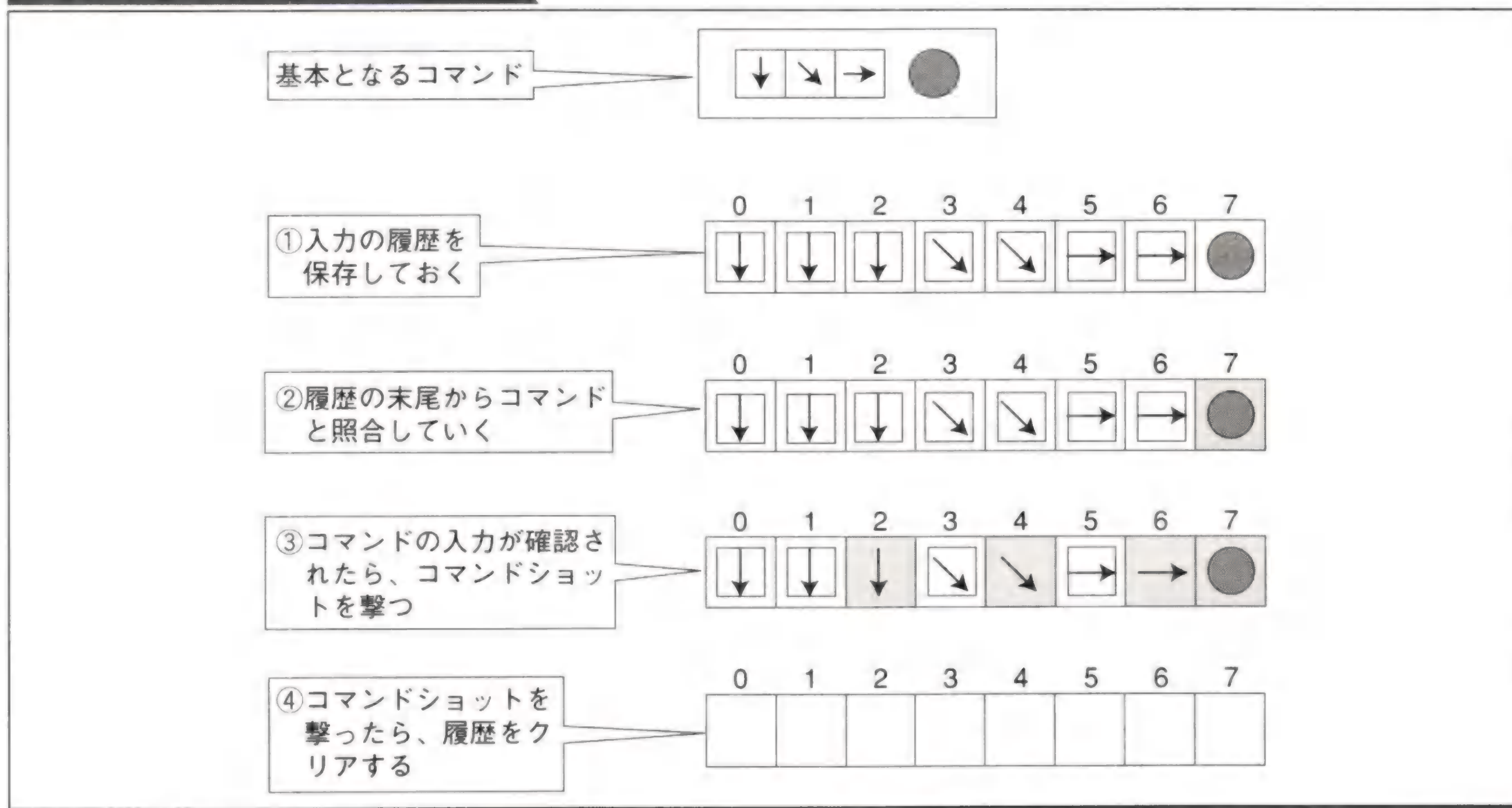
実際のプレイヤーの入力パターンはさまざまです。たとえば遅くコマンドを入力すると、1つの入力方向が複数回検出されるので、Fig. 4-15-②のような入力パターンになります。また、急いでコマンドを入力すると、Fig. 4-15-③のようにコマンドに関係ない入力方向が混じってし

まうこともよくあります。これと同様に、コマンドの途中にニュートラル（スティックをどの方向にも入力していない状態）が混じることもあります（Fig. 4-15-④）。

こういったコマンド入力の「ゆらぎ」をある程度は許したほうが、ゲームとしては遊びやすく面白くなります。そのためには、次のような仕組みでコマンド入力の判定を行います（Fig. 4-16）。

Fig. 4-16　コマンド入力の判定

基本となるコマンド　↓ ↘ → ●

	0	1	2	3	4	5	6	7
①入力の履歴を保存しておく	↓	↓	↓	↘	↘	→	→	●
②履歴の末尾からコマンドと照合していく	↓	↓	↓	↘	↘	→	→	●
③コマンドの入力が確認されたら、コマンドショットを撃つ	↓	↓	↓	↘	↘	→	→	●
④コマンドショットを撃ったら、履歴をクリアする								

まず、スティックやボタンの入力履歴を保存しておきます（Fig. 4-16-①）。保存する履歴の数は、コマンド入力を許す時間の長さに比例します。ここでは8個の履歴を保存していますが、実際にはもう少し多めに保存する必要があるでしょう。たとえば1/60秒周期のゲームで、コマンドを1秒以内に入力する場合には、最低60個の履歴を保存しなければなりません。

この入力履歴とコマンドとを照合して、コマンドが入力されたかどうかを調べます（Fig. 4-16-②）。照合は入力履歴の末尾、つまり新しい履歴から行います。図では履歴末尾のボタンとコマンド末尾のボタンとが一致したので、網かけで示しました。

同じ要領で入力履歴の末尾から先頭に向かってコマンドとの照合を行い、コマンドの入力が確認されたらコマンドショットを撃ちます（Fig. 4-16-③）。ショットを撃ったら、履歴はクリアしておきます（Fig. 4-16-④）。これは、コマンドショットが連続して発射されたり、前に入力したコマンドが次のコマンドに影響したりしないために必要な処置です。

Fig. 4-16のような方法を使えば、入力パターンにゆらぎがあっても、コマンド入力として受理されるようになります。間に余分な入力やニュートラルが混じってもよく、とにかく決められた時間内（入力履歴が保存されている期間内）にコマンドを入力すればよいのです。

■ 入力時間が異なるコマンドの扱い

コマンドの種類によって入力時間の制限が違う場合、入力履歴を保存する数は入力時間がもっとも長いコマンドに合わせます。入力時間が短いコマンドに関しては、入力履歴の一部（新しい履歴のいくつか）だけを使えばよいのです（Fig. 4-17）。

入力履歴を保存するためには、配列などを使います。この際、入力履歴をずっと保存していくと配列に無限の長さが必要になってしまうので、1つの配列を循環させて使うことになります（Fig. 4-18）。配列の末尾に達したら（Fig. 4-18-②）、次は配列の先頭に戻って再び履歴を保存します（Fig. 4-18-③）。この図では、最新の入力を保存する位置を網かけで示しました。

配列を循環させて入力履歴を保存する手法は、オプションを実現する方法に似ています（→P. 104）。シューティングゲームではこのように、配列（有限のバッファ）を循環させて使うことがよくあります。

Fig. 4-17 入力時間が異なるコマンドの扱い

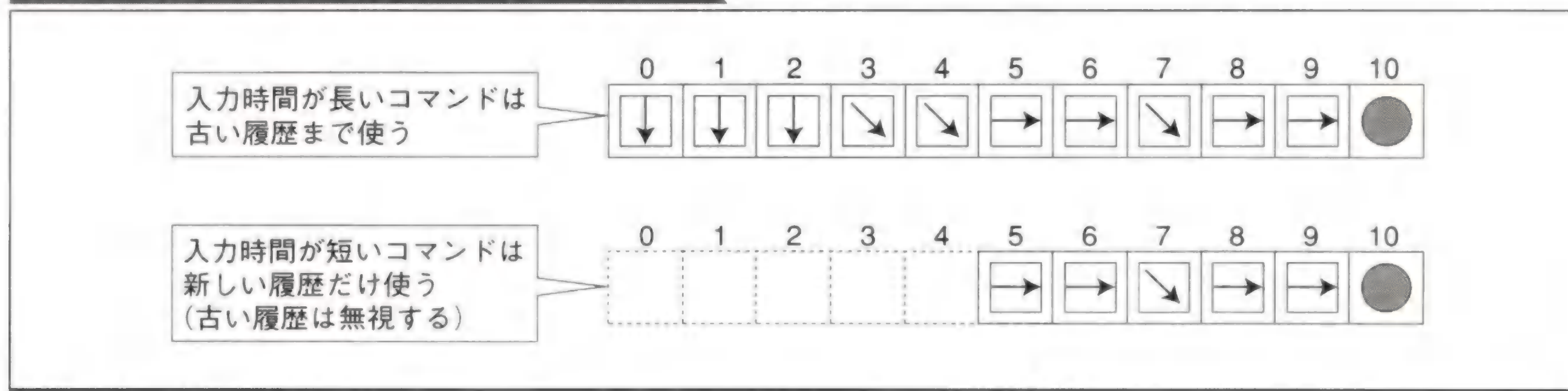
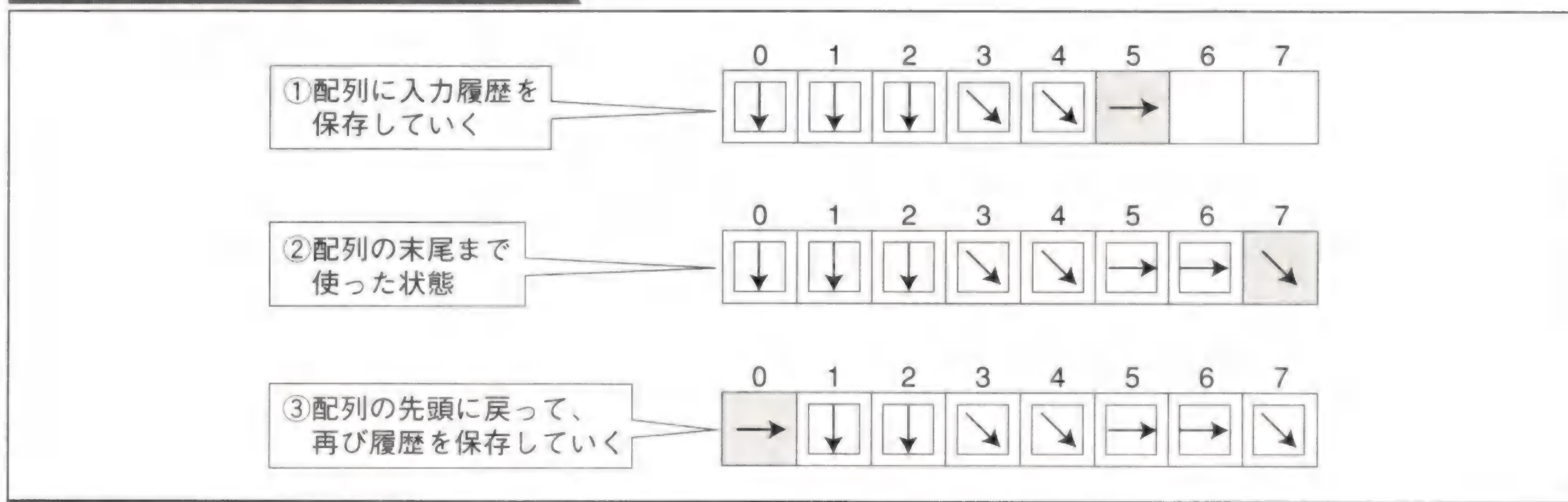



Fig. 4-18 配列を循環させて使う

サンプル

コマンドショット → P. 317

最後にコマンドショットの処理をList 4-8にまとめました。List 4-8-(1)は定数や変数の宣言、List 4-8-(2)は初期化用の関数、List 4-8-(3)はコマンドの判定を行う関数です。

List 4-8 コマンドショット

```
//================================================================ (1)
// スティックとボタンの状態を表す定数：
// 2つ以上の定数を組み合わせることもできる。
// たとえば「左上」は、
// 「上」と「左」の組み合わせ（UP|LEFT）で表す。
#define NONE      0  // 初期状態
#define NEUTRAL   1  // ニュートラル
#define UP        2  // 上
#define DOWN      4  // 下
#define LEFT      8  // 左
#define RIGHT    16  // 右
#define BUTTON   32  // ボタン

// 入力履歴の個数、入力履歴
#define NUM_HISTORY 30
static int History[NUM_HISTORY];

// コマンド
typedef struct {
    int Length;               // コマンドの長さ
    int Limit;                // 入力時間の制限
    int Input[NUM_HISTORY];   // 入力内容
} COMMAND_TYPE;
#define NUM_COMMAND 2
COMMAND_TYPE Command[NUM_COMMAND];

//================================================================ (2)
// 入力履歴とコマンドの初期化
void InitCommandShot()
{
    // 入力履歴の初期化
    for (int i=0; i<NUM_HISTORY; i++) History[i]=NONE;

    // コマンド1（波動拳コマンド）
    Command[0].Length=4;
    Command[0].Limit=30;
    Command[0].Input[0]=DOWN;
    Command[0].Input[1]=DOWN|RIGHT;
    Command[0].Input[2]=RIGHT;
    Command[0].Input[3]=BUTTON;

    // コマンド2（昇龍拳コマンド）
```

```
    Command[1].Length=4;
    Command[1].Limit=30;
    Command[1].Input[0]=RIGHT;
    Command[1].Input[1]=DOWN;
    Command[1].Input[2]=DOWN|RIGHT;
    Command[1].Input[3]=BUTTON;
}

//============================================================ (3)
// コマンドショットの判定を行う関数
void CommandShot(
    bool up, bool down,     // スティックの状態（上下左右）
    bool left, bool right,
    bool button             // ボタンの状態（押されたときtrue）
) {
    // 入力履歴の記録位置
    static int index=0;

    // 入力を履歴に記録する
    History[index]=
        (up?UP:0)|(down?DOWN:0)|
        (left?LEFT:0)|(right?RIGHT:0)|
        (button?BUTTON:0);

    // 各コマンドが入力されたかどうかを判定する
    int c, j, i;
    for (c=0; c<NUM_COMMAND; c++) {
        for (i=0; j=Command[c].Length-1; j>=0; j--) {
            for (i<Command[c].Limit; i++) {
                if (History[(index-i+NUM_HISTORY)
                    %NUM_HISTORY]==Command[c].Input[j]) break;
            }
            if (i==Command[c].Limit) break;
        }

        // コマンドの入力が確認できた：
        // コマンドショットを撃ち、履歴をクリアする。
        // 具体的な処理はComShot関数で行うとする。
        if (j==-1) {
            ComShot(j);
            for (int i=0; i<NUM_HISTORY; i++) History[i]=NONE;
        }
    }

    // 記録位置を更新する
    index=(index+1)%NUM_HISTORY;
}
```

ショットの移動

さまざまなショットについて解説してきましたが、ここで少し話題を変えて、ショットの移動について説明します。ショットの表示方法や移動方法にはゲームによって無数のバリエーションがありますが、ここで説明するのはすべてのショットの基本となる動きです。

一般に、ショットは自機の前方に出て、画面端に向かって飛んでいきます（Fig. 4-19）。ショットが画面枠から完全に出たところで、そのショットは消えます。

ショットが画面外に出たことの判定はFig. 4-20のように行います。これは弾が画面外に出たかどうかの判定と同じ方法です（→ P. 29）。ショットの左上座標を（x0, y0）、右下座標を（x1, y1）、画面枠の左上座標を（sx0, sy0）、右下座標を（sx1, sy1）とすると、ショットが完全に画面外にある条件は次のようになります。

(x1<=sx0 || sx1<=x0 || y1<=sy0 || sy1<=y0)

List 4-9は「ショットを一定速度で移動させて、画面外に出たら消す」という処理をまとめたプログラムです。

Fig. 4-19 ショットの動き

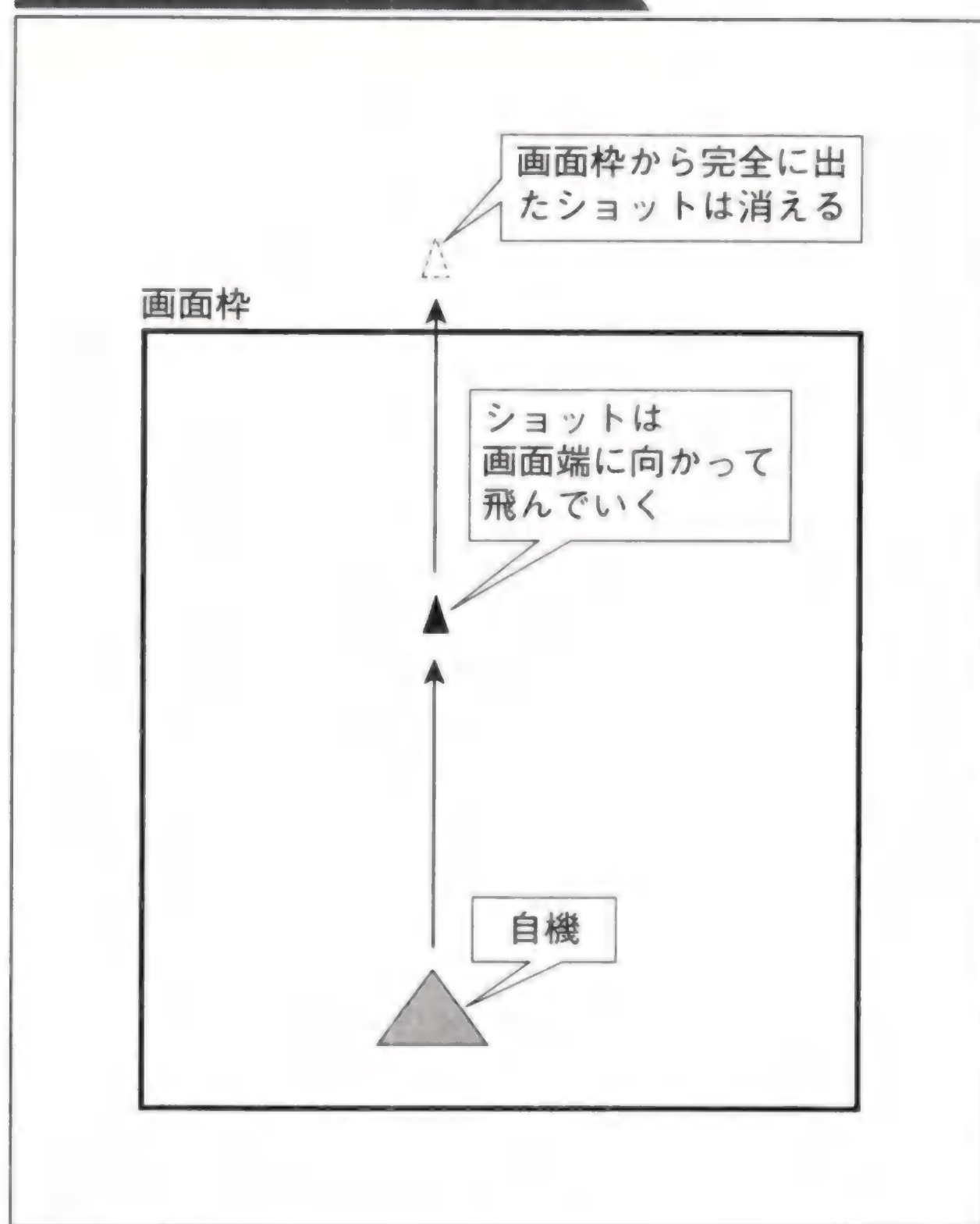
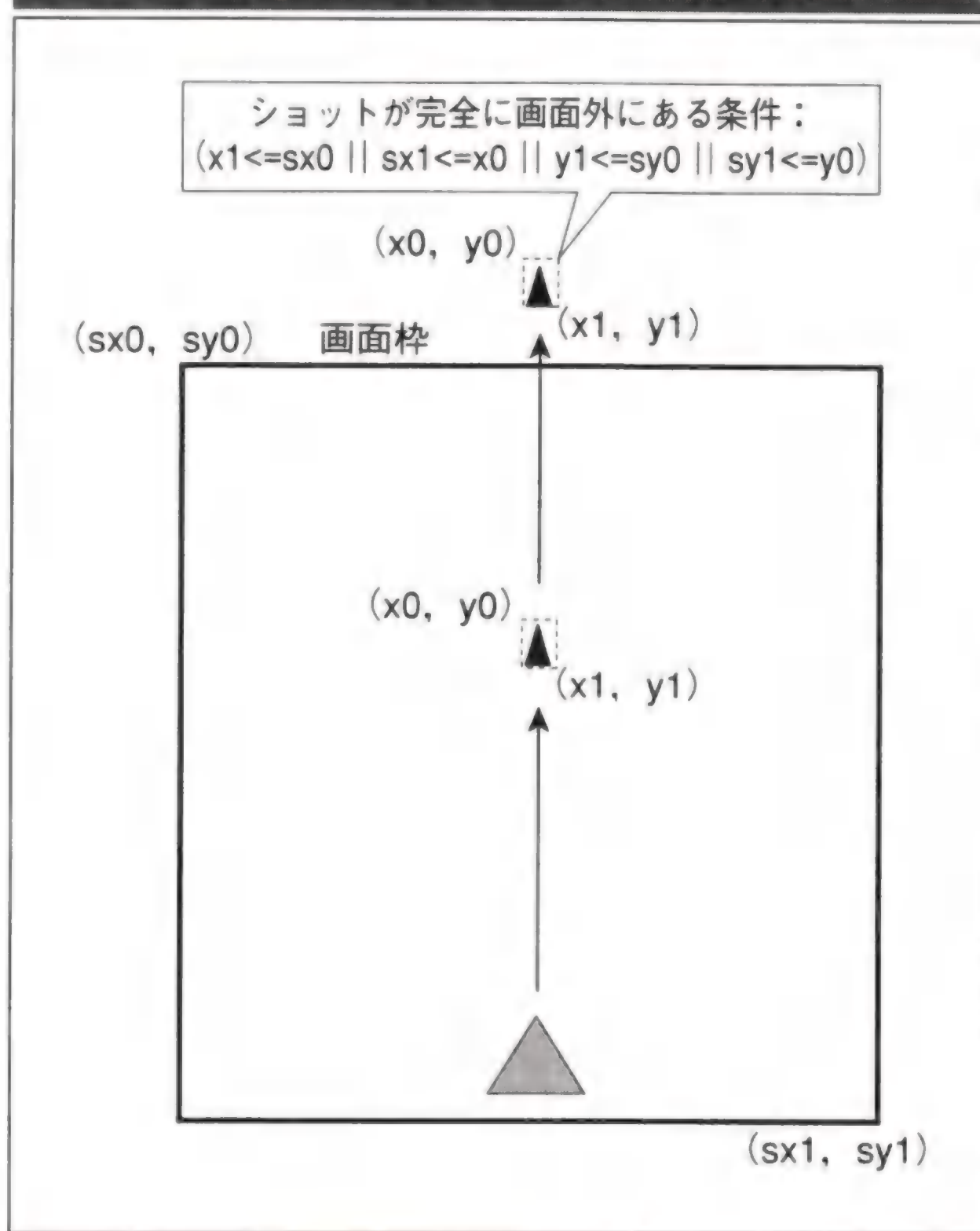



Fig. 4-20 ショットが画面外に出たことの判定

List 4-9 ショットの移動

```
void MoveShot(
    float& x0, float& y0,  // ショットの左上座標
    float& x1, float& y1,  // ショットの右下座標
    float vx, float vy,    // ショットの速度(X方向、Y方向)
    float sx0, float sy0,  // 画面枠の左上座標
    float sx1, float sy1   // 画面枠の右下座標
) {
    // ショットが画面外に出ていたら消す:
    // 具体的な処理はDeleteShot関数で行うとする。
    if (x1<=sx0 || sx1<=x0 || y1<=sy0 || sy1<=y0) DeleteShot();

    // ショットを移動する(座標を更新する)
    x0+=vx; y0+=vy;
    x1+=vx; y1+=vy;
}
```

ショットの当たり判定処理

次はショットの当たり判定処理について考えます。ショットが当たる対象は主に敵と地形です。破壊可能な敵や地形に当たったときには、ショットは所定の攻撃力で対象の耐久力を削ります(Fig. 4-21)。敵や地形は耐久力がなくなると破壊されます。破壊不可能な敵や地形に当たったときには、ショットは消えます。

ショットと対象との当たり判定処理はFig. 4-22のように行います。ショットの当たり判定の左上座標を(x0, y0)、右下座標を(x1, y1)、対象の当たり判定の左上座標を(ex0, ey0)、右下座標を(ex1, ey1)とすると、ショットが対象に当たる条件は次のどちらかになります。

!(x1<=ox0 || ox1<=x0 || y1<=oy0 || oy1<=y0)

(ox0<x1 && x0<ox1 && oy0<y1 && y0<oy1)

ショットの当たり判定処理をまとめたのがList 4-10です。対象に命中したあとのショットを消さないと、敵を貫通するショットになります。ショットの側にも耐久力を設ければ、「耐久力が0になるまでは貫通するショット」を作ることもできます。

Fig. 4-21 ショットが当たる対象

破壊可能な敵や地形にショットが命中
→ショットの攻撃力で対象の耐久力を削る

破壊不可能な敵や地形にショットが命中
→ショットは消える

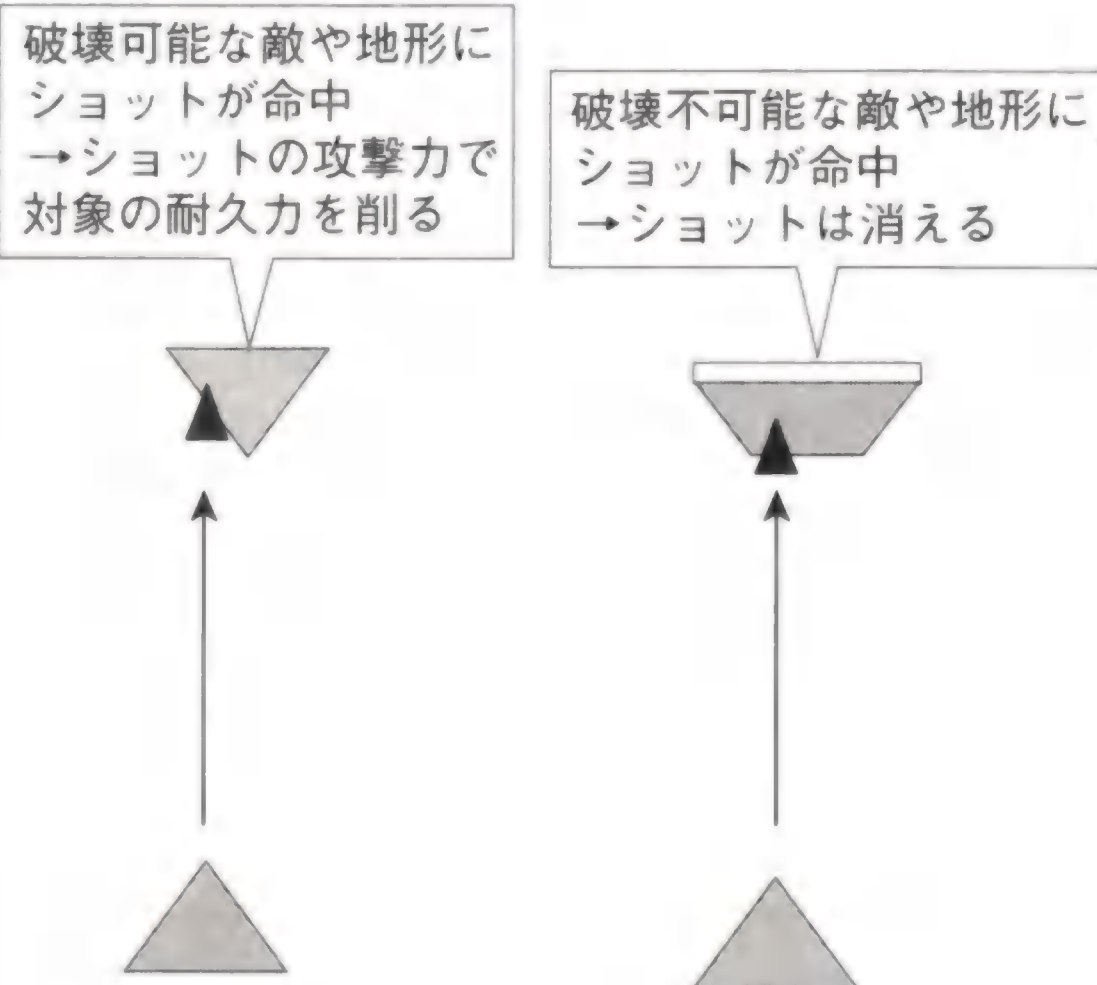
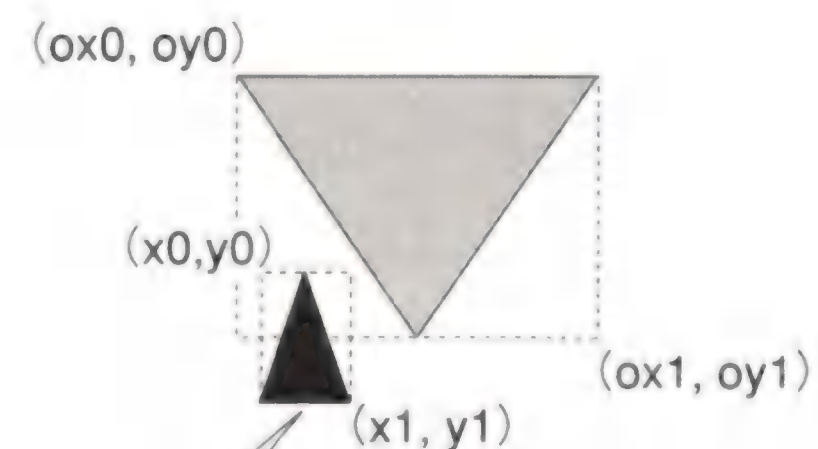



Fig. 4-22 ショットと対象との当たり判定処理

(ox0, oy0)
(x0,y0)
(ox1, oy1)
(x1, y1)

ショットが対象に当たる条件：
`!(x1<=ox0 || ox1<=x0 || y1<=oy0 || oy1<=y0)`
または
`(ox0<x1 && x0<ox1 && oy0<y1 && y0<oy1)`

List 4-10 ショットの当たり判定処理

```
void HitShot(
    float x0, float y0,     // ショットの当たり判定の左上座標
    float x1, float y1,     // ショットの当たり判定の右下座標
    float attack,           // ショットの攻撃力
    float ox0, float oy0,   // 対象の当たり判定の左上座標
    float ox1, float oy1,   // 対象の当たり判定の右下座標
    float endurance,        // 対象の耐久力
    bool invincible         // 対象が破壊不可能かどうか
) {
    // ショットが対象に当たった場合
    if (ox0<x1 && x0<ox1 && oy0<y1 && y0<oy1) {

        // 対象が無敵でない場合：
        // ショットの攻撃力で対象の耐久力を削り、
        // 耐久力が0になったら対象を消す。
        // 具体的な処理はDeleteOpponent関数で行うとする。
        if (!invincible) {
            endurance-=attack;
            if (endurance<0) DeleteOpponent();
        }

        // ショットを消す：
        // 具体的な処理はDeleteShot関数で行うとする。
        // ここでショットを消さないと、貫通弾になる。
        DeleteShot();
    }
}
```

敵との距離によるショットの威力の違い

ショットを敵の近くで当てるほど、ショットの威力が強くなるというゲームもあります(Fig. 4-23)。敵に近づくのは危険ですが、近づくほどショットが強くなるとなれば、プレイヤーは積極的に敵に近づいて、敵を早く倒そうとするでしょう。ここに絶妙な緊張感が生まれて、ゲームが面白くなります。

このように敵との距離によってショットの威力を変えるには、ショットの飛行距離によって威力を弱めてやります(Fig. 4-24)。敵に近いときにはショットの飛行距離が短くなるので、威力は強くなります。敵から遠いときには飛行距離が長くなるので、威力は弱められます。

飛行距離によってショットの威力を変える処理をまとめたのがList 4-11です。発射された直後のショットは威力が最強で、飛行するにつれて威力が弱まっていきます。

Fig. 4-23 敵との距離によるショットの威力の違い

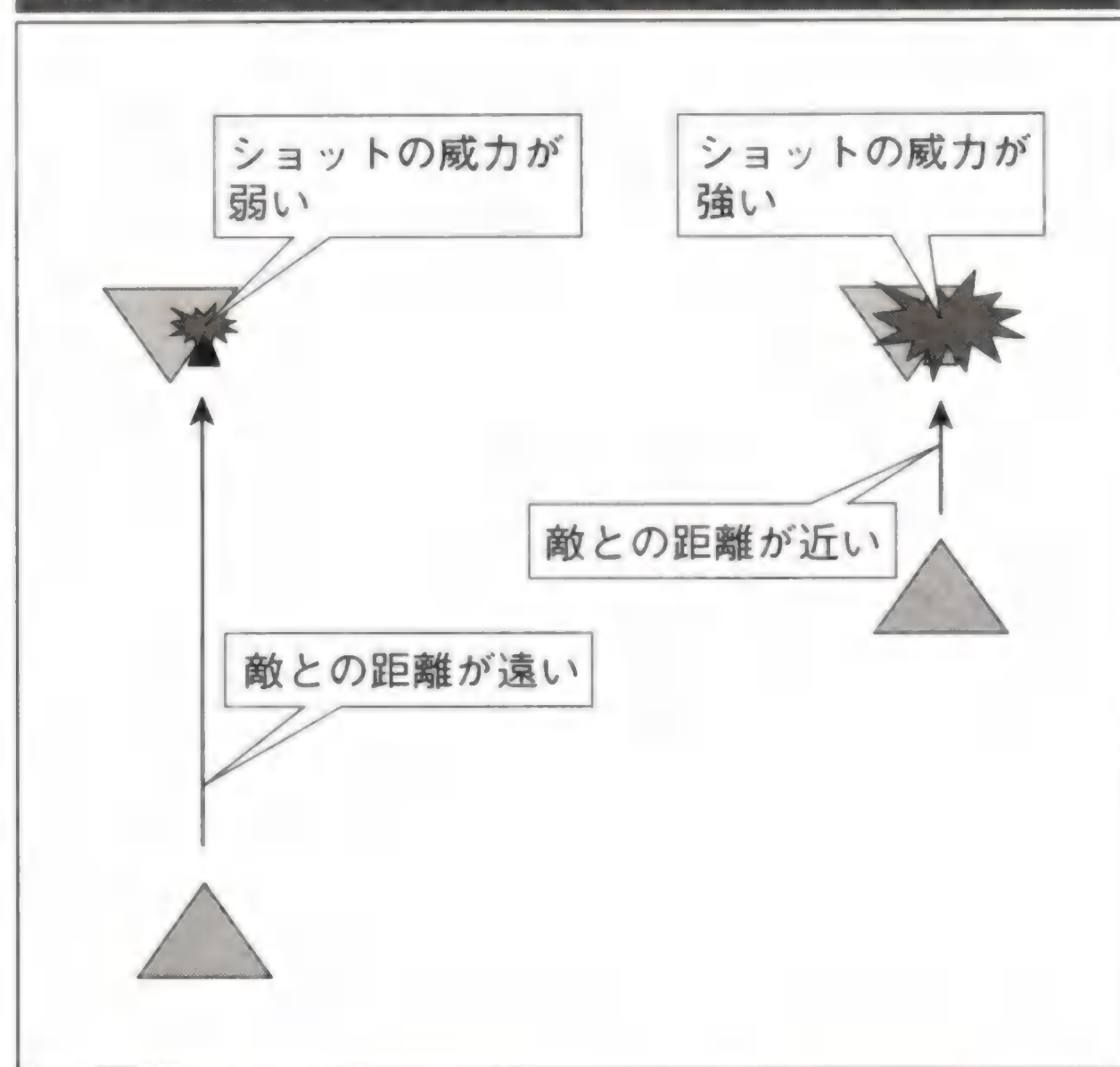
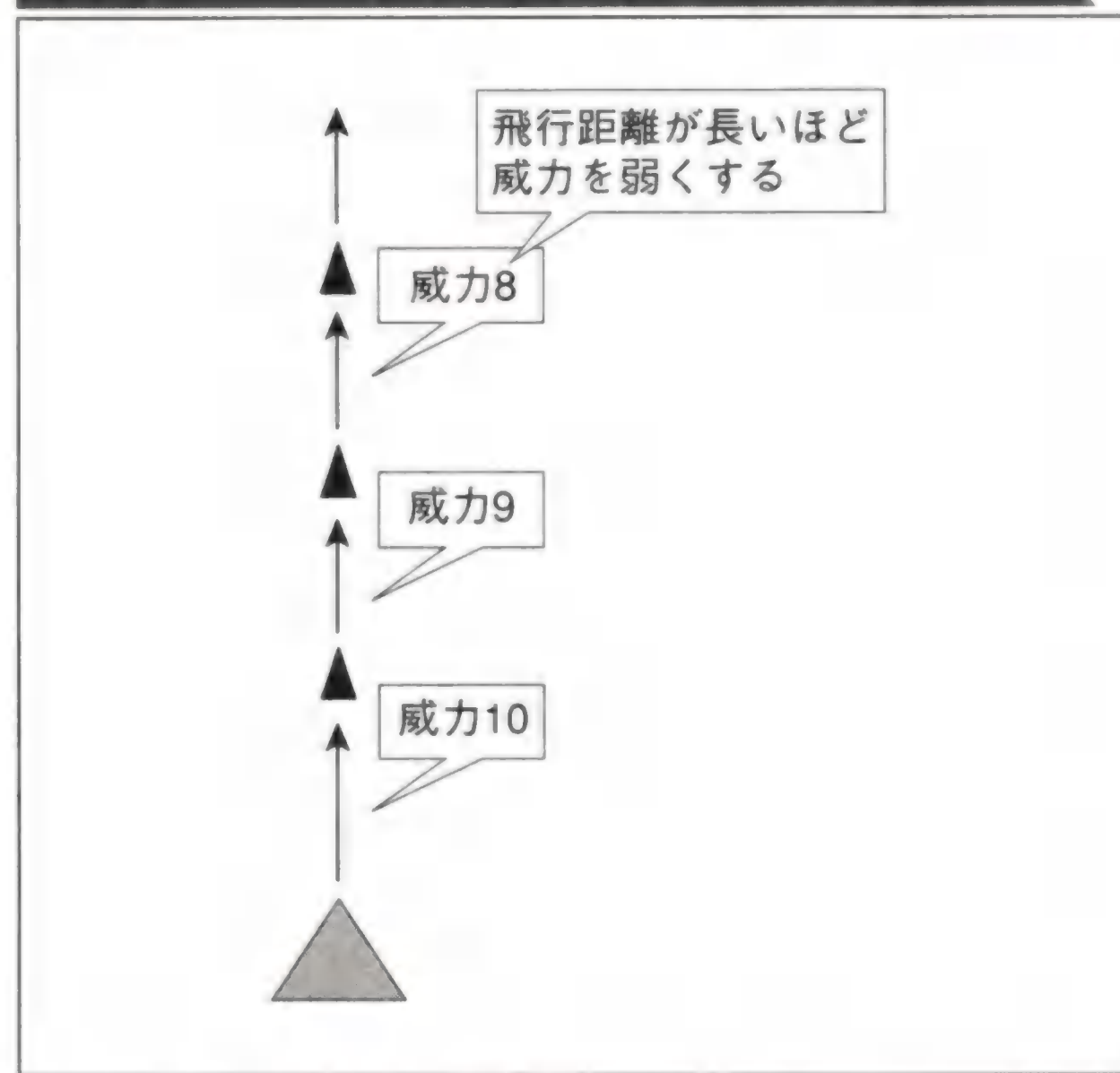



Fig. 4-24 ショットの飛行距離と威力の関係

List 4-11 敵との距離によるショットの威力の違い

```
void ShotPower(
    float& x, float& y,  // ショットの座標
    float vx, float vy,  // ショットの速度
    float& power,        // ショットの威力
    float attenuation    // 威力が減衰する度合い
) {
    // ショットを移動させる
    x+=vx; y+=vy;

    // 移動するたびにショットの威力を弱める
```

```
    power-=attenuation;
}
```

照準を使った爆撃

自機の前に表示された照準で目標を狙い、そこに爆弾を落とす攻撃手段です。これはかの有名な「ゼビウス (→ P. 329)」が採用したルールで、その後のシューティングゲームに多大な影響を与えました。ゼビウスでは空中の敵を撃つショット「ザッパー」と、地上の敵を爆撃する「ブラスター」を使い分けます。照準を使った爆撃は「ブラスター」です。照準を使った爆撃にはさまざまなバリエーションが考えられますが、ここではゼビウスの方式に基づいて説明することにします。

まず、照準は自機の前方に表示されます (Fig. 4-25)。自機と照準との位置関係は常に変わらず、自機が動くと照準もいっしょに動きます。照準が地上物 (地上にいる敵) に重なると、照準の端が点滅して、爆撃が可能だということをプレイヤーに示します (Fig. 4-26)。

Fig. 4-25 照準、自機、地上物

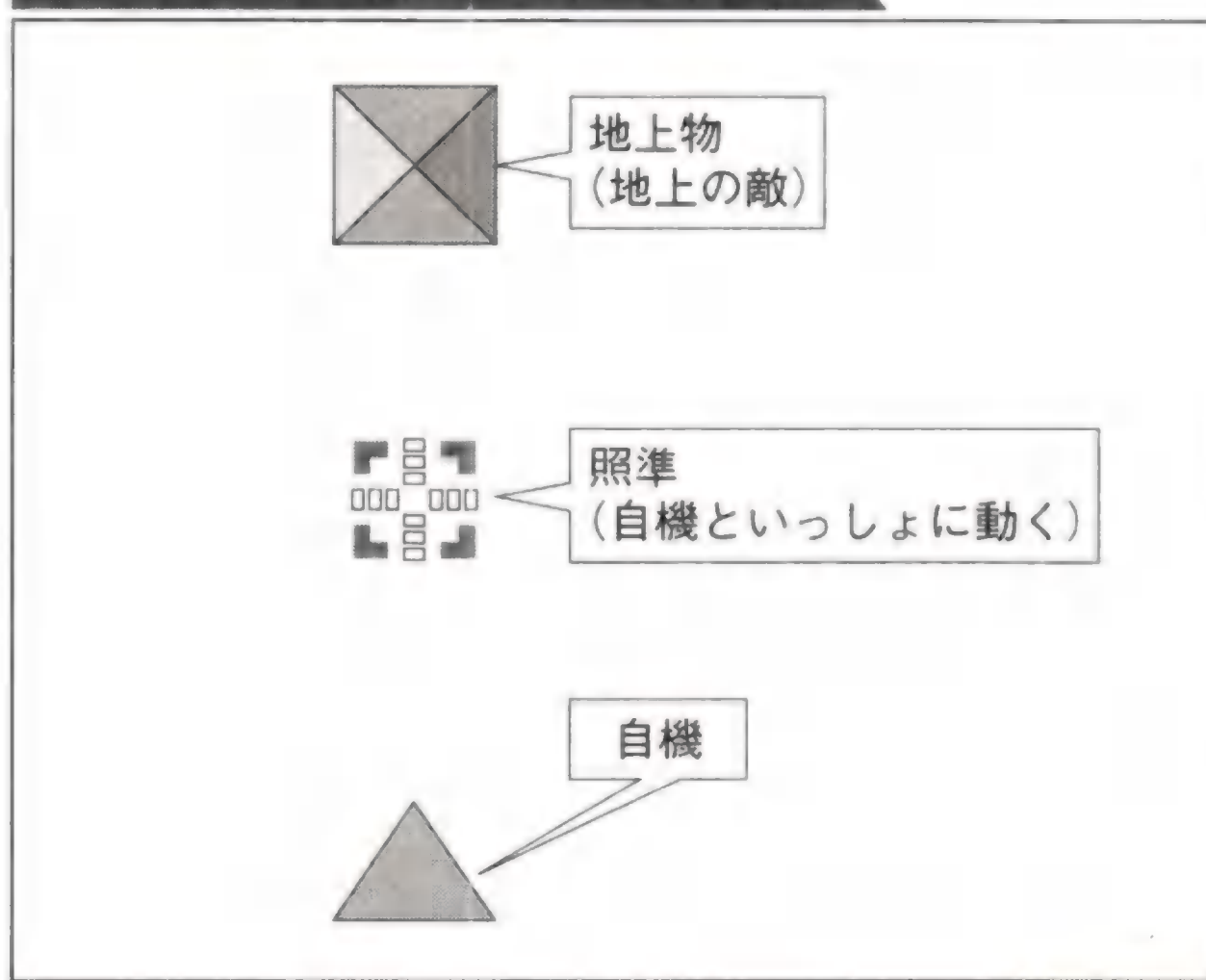
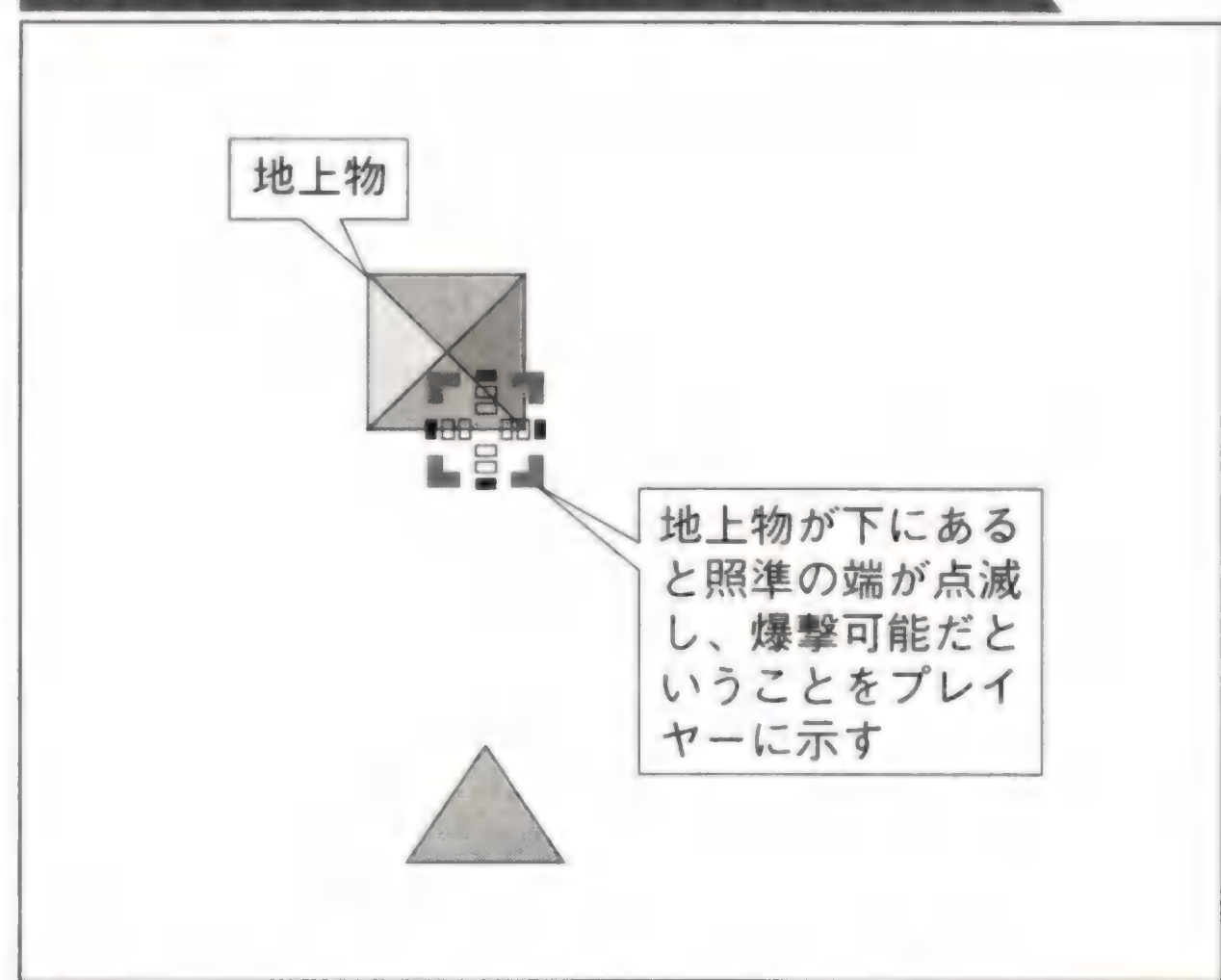



Fig. 4-26 地上物が照準の下にあるとき

実際に爆撃を行うと、着弾点のマークが表示され、自機から爆弾が発射されます (Fig. 4-27)。この状態になったら、あとは爆弾が自動的に目標に当たってくれるので、自機はほかの場所に移動してもかまいません (Fig. 4-28)。

ちなみに「ゼビウス」の場合、ブラスターは一度に1発ずつしか撃てないので、撃ったブラスターが着弾するまで次のブラスターは撃てません。次のブラスターが撃てない間は照準の形が十字型になり、撃てるようになると元の形 (ファインダー型) に戻ります。こういったプレイヤーへの細かな情報提示のうまさも、ゼビウスを傑作たらしめた要因の1つでしょう。

Fig. 4-27 爆撃の実行

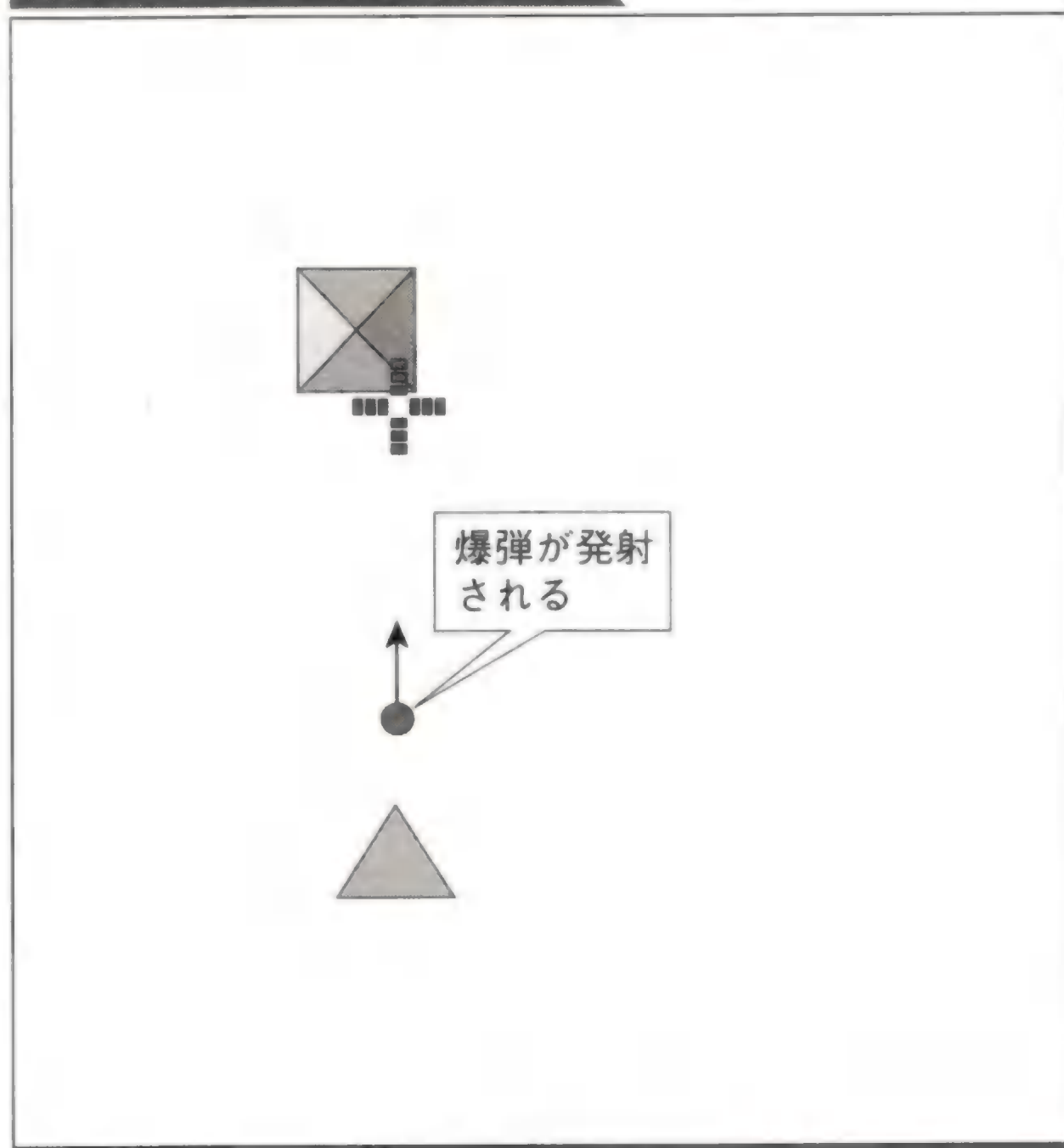
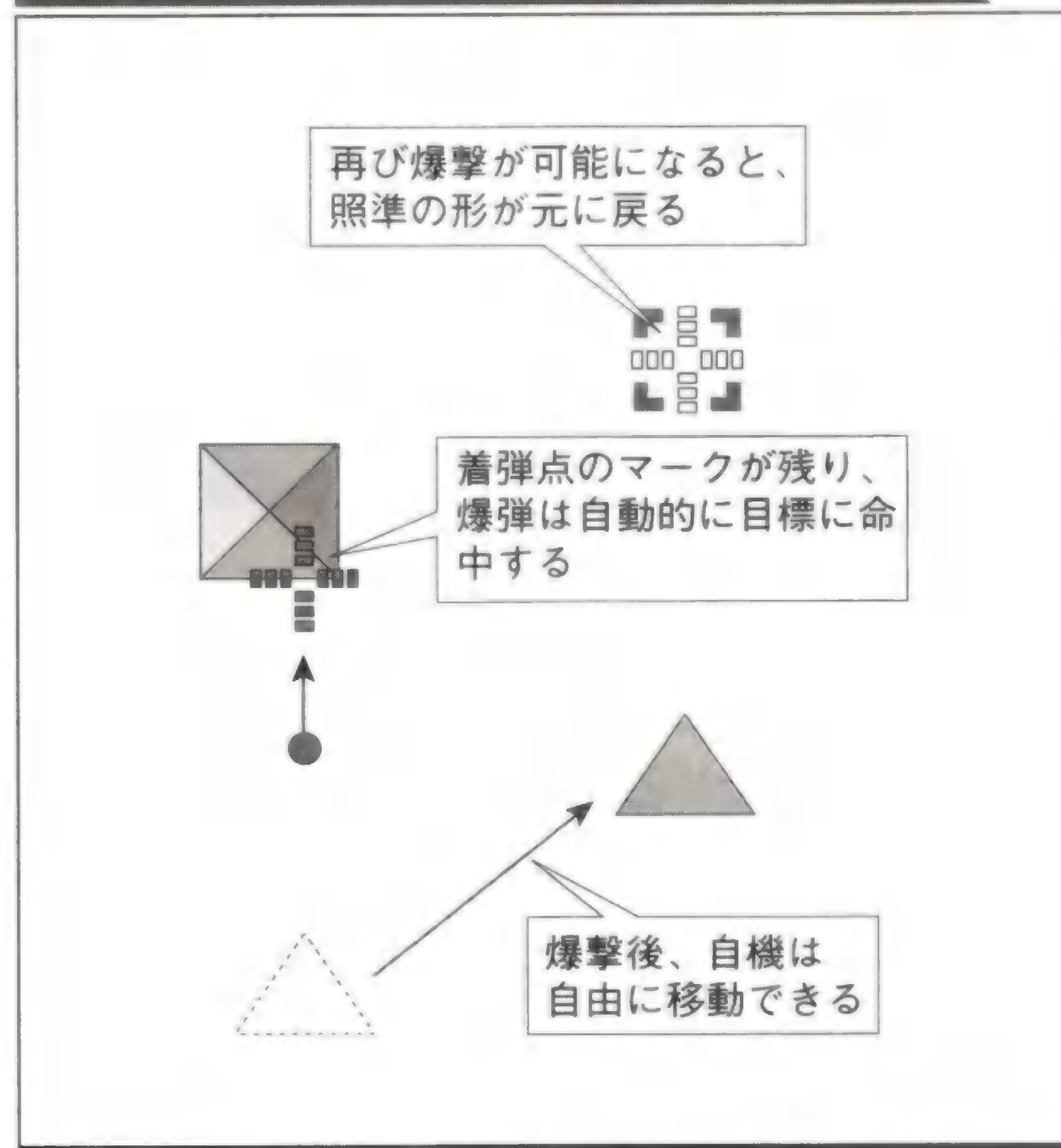



Fig. 4-28 自機は爆撃後に移動してもよい

照準を使った爆撃をプログラムにまとめると、List 4-12のようになります。ポイントは爆撃中とそれ以外のときとで移動や描画の処理を分岐させることです。

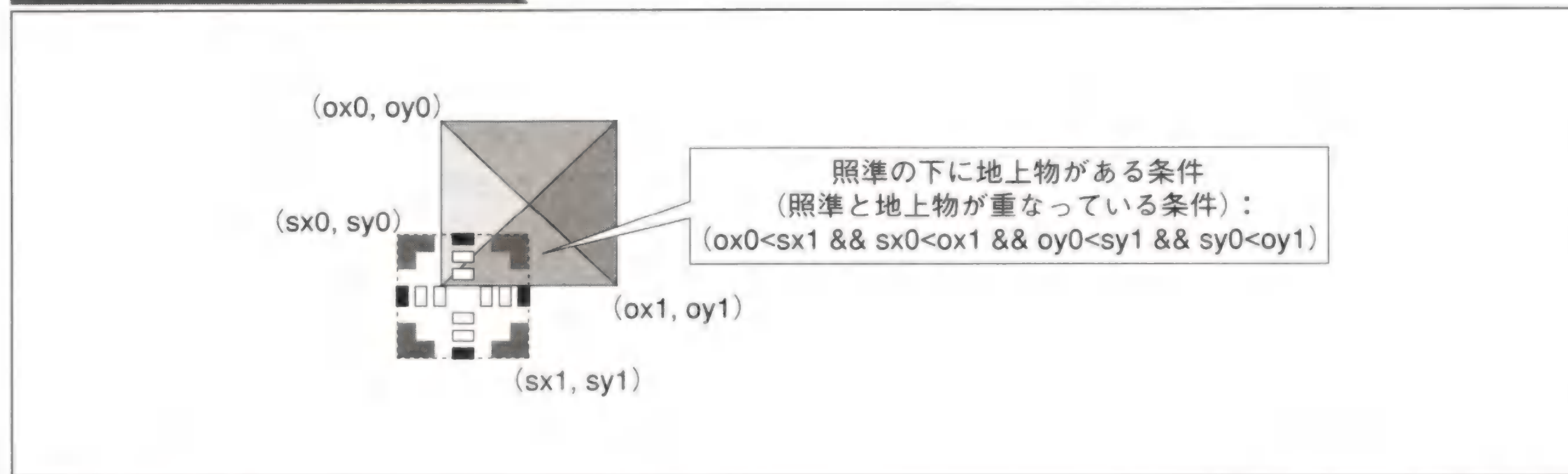
なお、照準と地上物との当たり判定処理を行えば、下に地上物があるときに照準を点滅させることもできます。

照準の左上座標を (sx0, sy0)、右下座標を (sx1, sy1)、地上物の左上座標を (ox0, oy0)、右下座標を (ox1, oy1) とすると、照準の下に地上物がある条件は次のようになります (Fig. 4-29)。

(ox0<sx1 && sx0<ox1 && oy0<sy1 && sy0<oy1)

サンプル

照準を使った爆撃 → P. 317

Fig. 4-29 照準を点滅させる

List 4-12 照準を使った爆撃

```
void SightedBomb(
    float x, float y,          // 自機の座標
    float sx, float sy,        // 照準の座標
    float bvx, float bvy,      // 爆弾の速度
    bool button                // ボタンの状態(押されたときtrue)
) {
    // 爆弾の状態
    static bool bombing=false;  // 爆撃中かどうか
    static float bx, by;        // 爆弾の座標
    static float tx, ty;        // 着弾点の座標

    // 爆撃中ではないとき:
    // ボタンが押されたら爆撃する。爆弾の初期座標と着弾点の座標を設定する。
    if (!bombing) {
        if (button) {
            bombing=true;
            bx=x; by=y;
            tx=sx; ty=sy;
        }
    }

    // 爆撃中のとき:
    // 爆弾を移動させる。爆弾が着弾点に到達したら爆発させる。
    // 爆発の具体的な処理はExplode関数で行うとする。
    else {
        bx+=bvx; by+=bvy;
        if (bx==tx && by==ty) {
            Explode();
            bombing=false;
        }
    }

    // 自機と照準を描く:
    // 具体的な処理はDrawMyShip関数とDrawScope関数で行うとする。
    // 照準と地上物との当たり判定処理を行えば、
    // 照準の端を点滅させることもできる。
    DrawMyShip(x, y);
    DrawScope(sx, sy);

    // 爆撃中の場合には着弾点と爆弾を描く:
    // 具体的な処理はDrawTarget関数とDrawBomb関数で行うとする。
    if (bombing) {
        DrawTarget(tx, ty);
        DrawBomb(bx, by);
    }
}
```

ロックオンレーザー

照準で敵をロック（補足）し、ロックした敵に向かって誘導レーザーを放つという武器です。「ゼビウス（→ P. 329）」以来、照準を使って地上物を攻撃する形式のゲームは多くありましたが、同じように照準を使いつつも、まったく別のオリジナル武器にまで発展させたのが「レイフォース（→ P. 335）」シリーズのロックオンレーザーだといえます。ロックオンレーザーは使い勝手がよく、見た目にも非常にインパクトがある武器なので、2Dゲームだけではなく「パンツァードラグーン（→ P. 332）」などの3Dゲームにも採用されています。

ロックオンレーザーでは、照準を使って敵をロックします（Fig. 4-30）。照準はゼビウスと同じように自機の前方にあり、自機といっしょに移動します。この照準を動かして敵の上をなぞると、なぞられた敵をロックします（Fig. 4-31）。

Fig. 4-30 ロックオンレーザーの照準

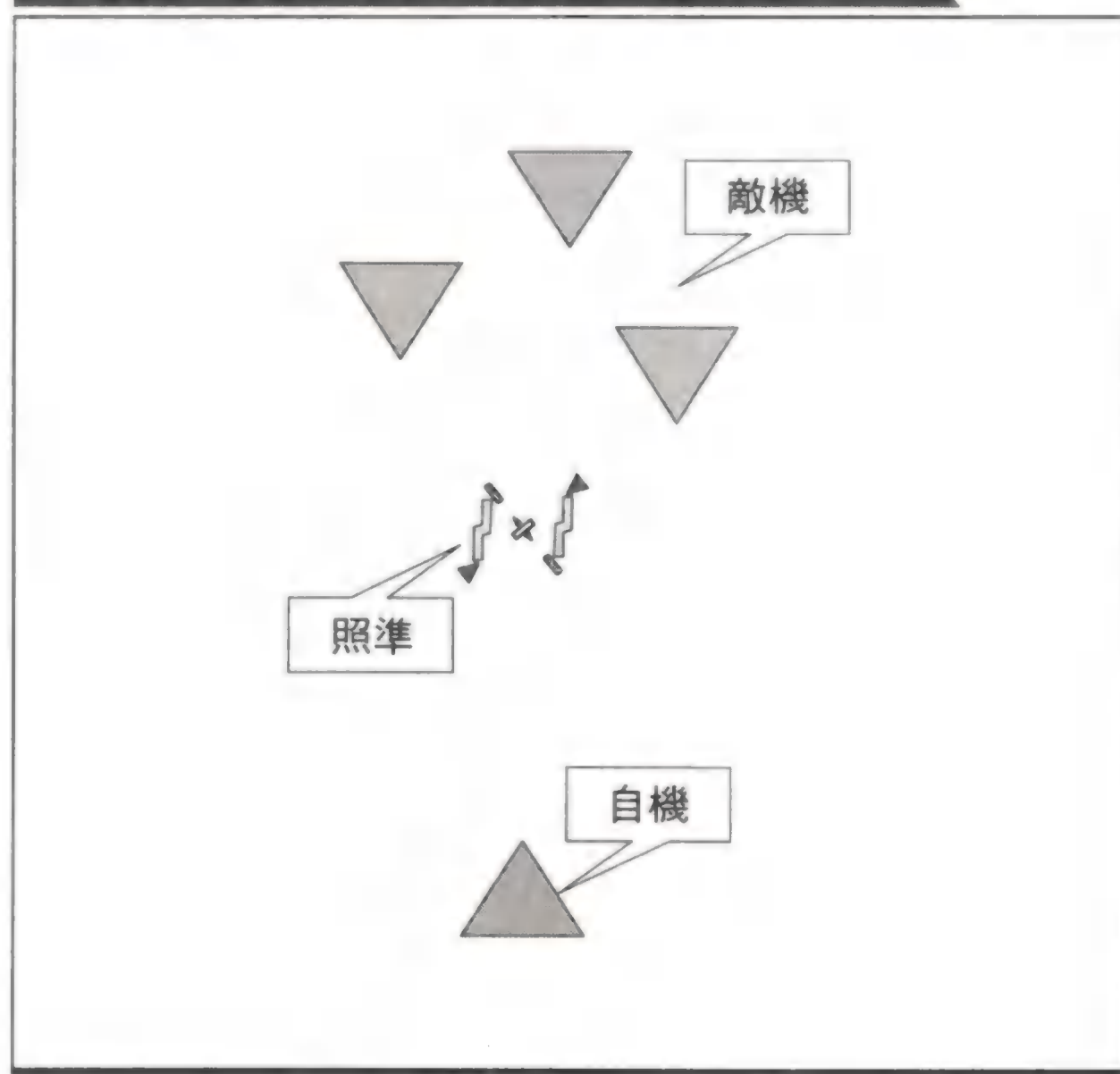
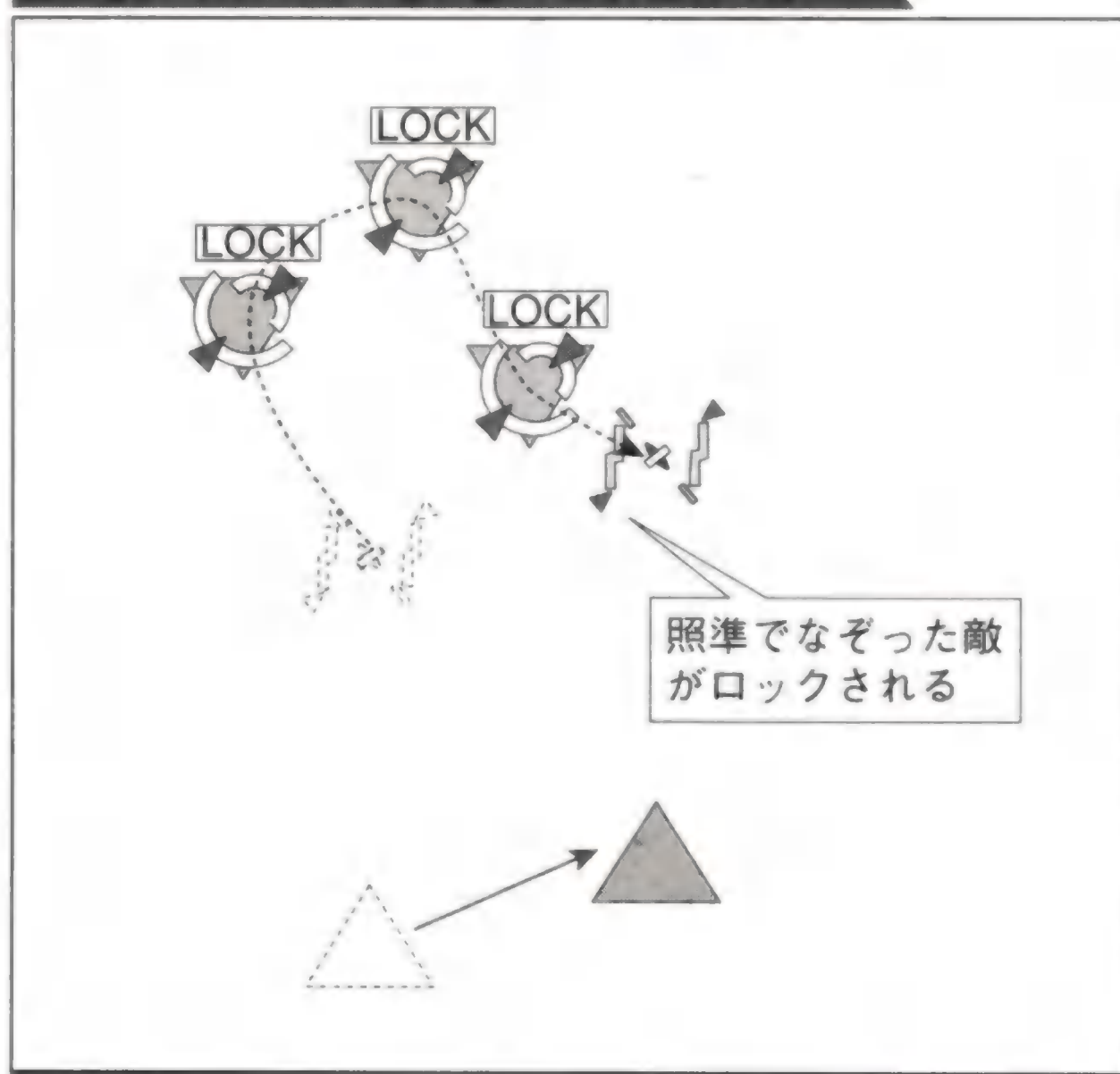
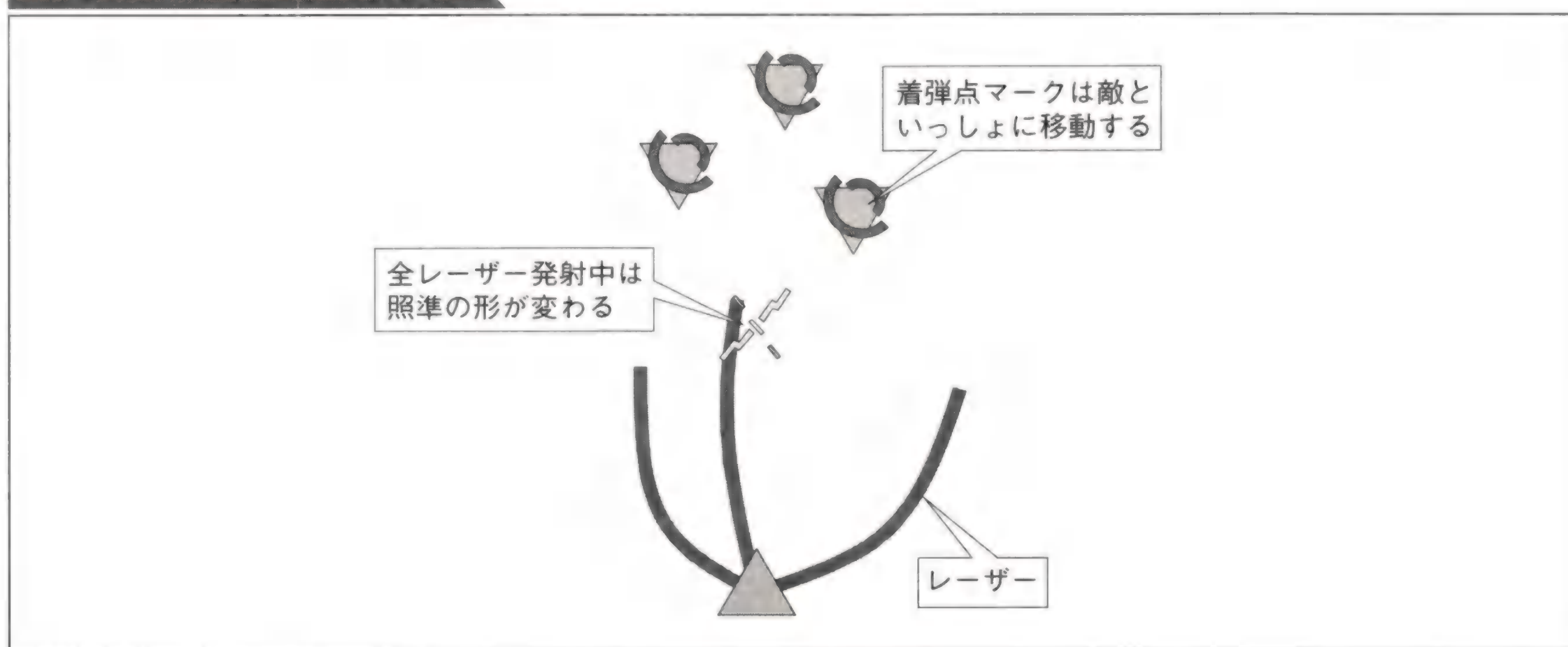



Fig. 4-31 照準の移動と敵のロック

ロックした敵にはロックマークがつきます。照準が移動して敵の上から外れても、しばらくの間はロックは外れません。敵を連続してなぞれば、複数の敵をまとめてロックすることもできます。「レイフォース」の場合、同時ロックが可能な敵の最大数は8です。レイフォースの後継作品（「レイストーム（→ P. 335）」と「レイクライシス（→ P. 334）」）では自機の種類によってロック可能な敵の最大数が異なります。

敵をロックした状態でボタンを押すと、ロックした敵に向かってレーザーが発射されます（Fig. 4-32）。レーザーが発射されるとロックマークは形が変わり（ここでは「着弾点マーク」と呼びます）、敵といっしょに移動します。「レイフォース」の場合、全レーザーの発射中は照準の形を変えて、プレイヤーにそれ以上レーザーが撃てないことを知らせます。レーザーが命中して再びレーザーが使用可能になると、照準の形は元に戻ります。

Fig. 4-32 レーザーの発射

着弾点マークは敵といっしょに移動する
全レーザー発射中は照準の形が変わる
レーザー

■ レーザーの動き

レーザーは「誘導レーザー」(→ P. 46) のような曲線を描きながら飛び、敵 (着弾点マーク) を目指します。敵が放つ誘導レーザーとは違って、目標が画面内にいるかぎり、レーザーは必ず命中します (Fig. 4-33)。敵が画面外に出て消えた場合だけは、レーザーが当たらないことがあります。

ロックオンレーザーの動きは基本的には誘導レーザーと同じですが、目標に必ず (あるいはほとんどの場合) 命中することが特徴です。目標に必ず命中させるには、ロックオンレーザーの旋回角度を高めにしておくかあるいは旋回角度がだんだん高くなるようにします (Fig. 4-34)。

Fig. 4-33 レーザーの命中

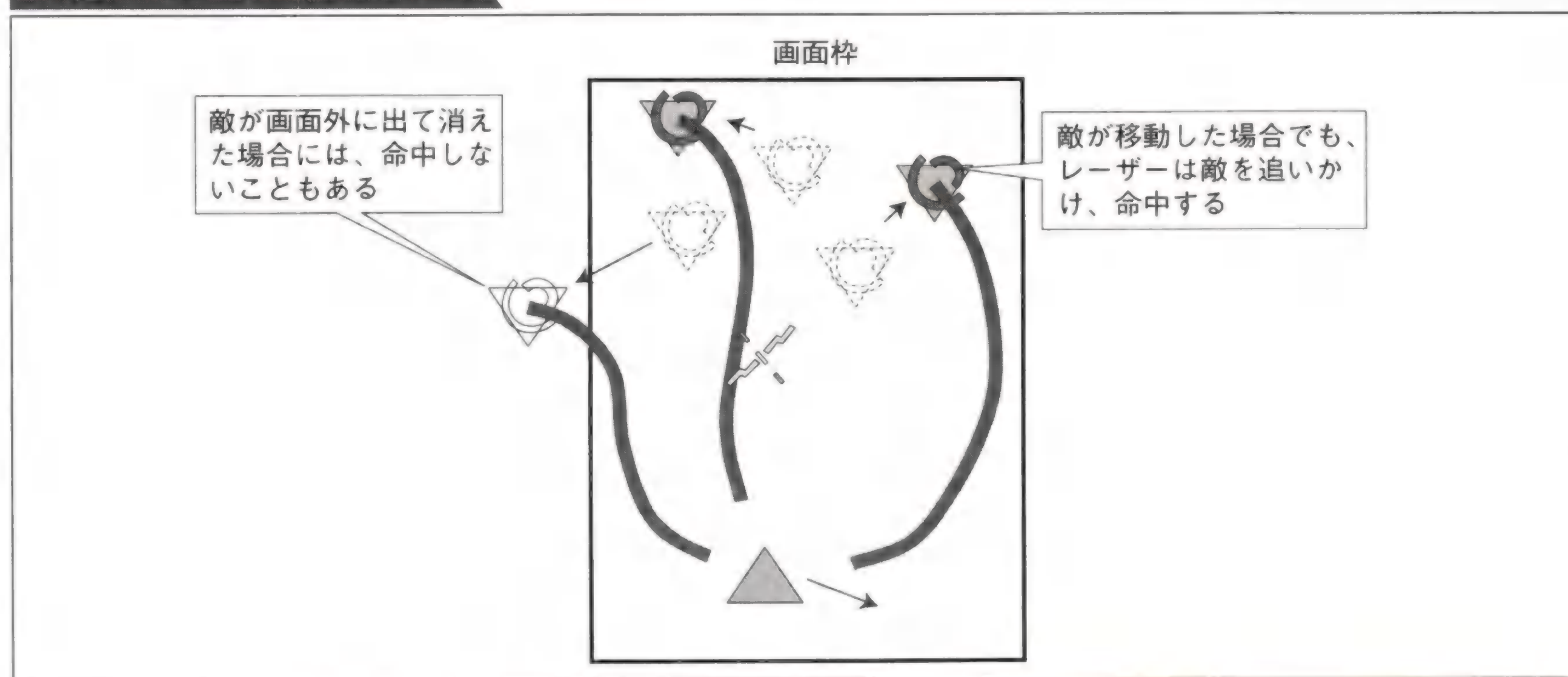
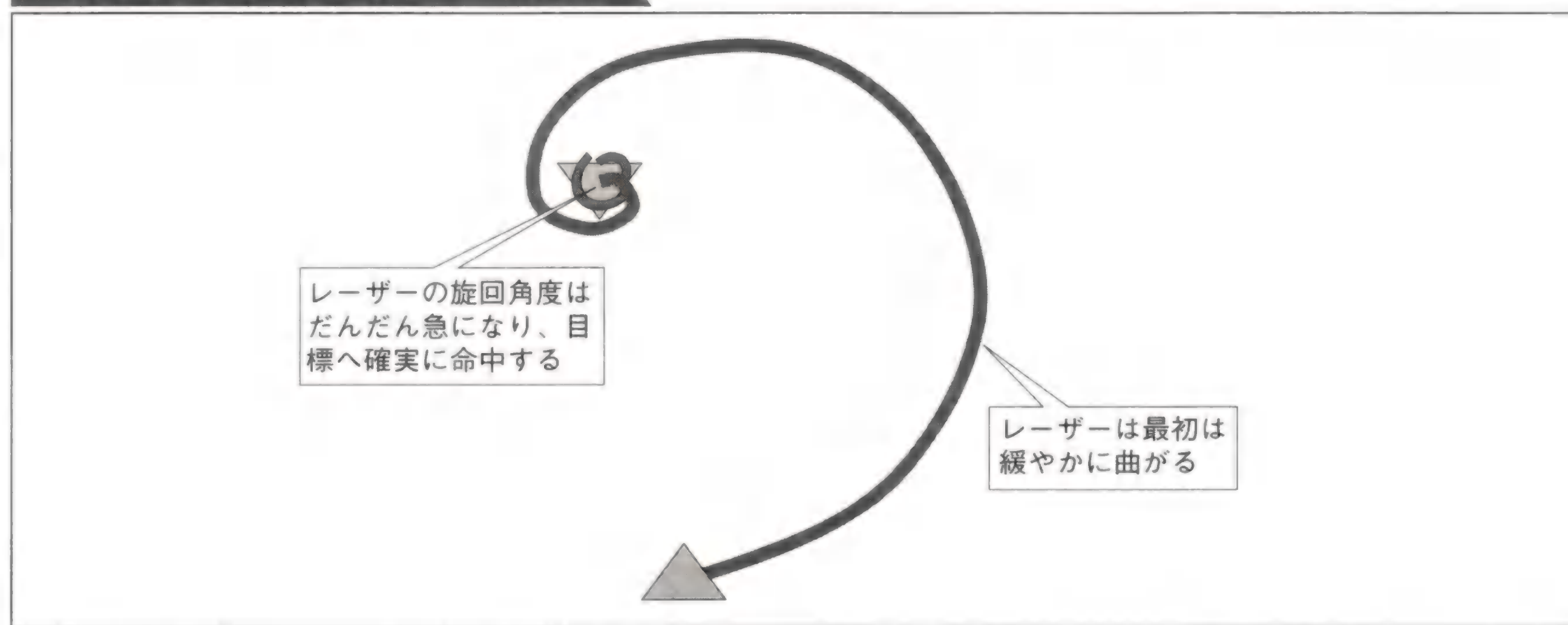
画面枠
敵が画面外に出て消えた場合には、命中しないこともある
敵が移動した場合でも、レーザーは敵を追いかけ、命中する

発射された直後のレーザーは緩やかに旋回していても、発射から時間が経つにつれて旋回角度の上限が上がって、急角度で曲がるようにします。これならば、最初は敵の周りを曲がっていたレーザーも、最後には確実に敵をとらえることができます。ただし、レーザーの旋回角度や速度の調節が悪いと「当たらないロックオンレーザー」になってしまうので、パラメータを調整する必要はあります。

Fig. 4-34 レーザーの旋回角度の変化

ロックオンレーザーの発射に関する処理をプログラムにまとめると、List 4-13のようになります。ポイントは、敵の構造体なりクラスなりに「その敵がロックされているかどうか」を表すフラグを持たせることです。そして、照準と敵との当たり判定処理を行って、敵がまだロックされていないときにはロックオンします（ロックマークをつけます）。ボタンが押されたら、ロックオンされた（ロックマークがついた）すべての敵に向かってレーザーを発射します。

レーザーの移動は「誘導弾（旋回角度を制限したもの）」と同じアルゴリズムです（→ P. 39）。ただし、速度や旋回角度を調整することで、敵へ確実に命中するようにします。旋回角度を変化させる場合には、時間とともに旋回角度の上限をじりじりと上げてやれば、レーザーはしだいに急角度で旋回するようになるので、敵へ確実に命中させることができます。

サンプル

● ロックオンレーザー　→ P. 317

List 4-13 ロックオンレーザーの発射

```
// 敵を表す構造体
typedef struct {
    bool Locked;  // ロックされているかどうか
    float X, Y;   // 座標
} ENEMY_TYPE;
#define MAX_ENEMIES 100
static ENEMY_TYPE Enemy[MAX_ENEMIES];

// ロックオンの使用可能数
static int AvailLocks=8;

// ロックオンレーザーの発射
void LockOnLaser(
    float sx, float sy,  // 照準の座標
    bool button          // ボタンの状態(押されたときtrue)
) {
    // ロックオンの使用可能数が1以上のとき:
    // すべての敵と照準との当たり判定処理を行い、
    // 照準と重なった敵にはロックマークをつける。
    // 当たり判定の具体的な処理はHit関数で行うとする。
    for (int i=0; AvailLocks>0 && i<MAX_ENEMIES; i++) {
        if (!Enemy[i].Locked && Hit(sx, sy, Enemy[i])) {
            Enemy[i].Locked=true;

            // ロックオンの使用可能数を減らす:
            // 敵にレーザーが命中したら、使用可能数を増やす。
            AvailLocks--;
        }
    }

    // ボタンが押された場合:
    // ロックマークがついたすべての敵に向けてレーザーを発射する。
    // 発射の具体的な処理はShootLockOnLaser関数で行うとする。
    if (button) {
        for (int i=0; i<MAX_ENEMIES; i++) {
            if (Enemy[i].Locked) ShootLockOnLaser(Enemy[i]);
        }
    }
}
```

地形に沿って飛ぶミサイル

壁や床といった地形に沿って飛ぶミサイルです（Fig. 4-35）。地形があるゲーム（横スクロールゲームが多い）で、地上物（地上の敵）を攻撃するためなどに使います。この方式のミサイルを使った代表的なゲームは「グラディウス（→ P. 326）」シリーズです。

こういったミサイルを実現するには、ミサイルの速度を水平方向と垂直方向に分けて考えるとうまくいきます（Fig. 4-36）。ミサイルが斜めに落ちていくのは、ミサイルが水平方向に進んでいるのと同時に垂直方向に落ちているからです。水平と垂直の動きが合成されて、斜め方向の動きになります。

ミサイルを地形に沿って飛ばすためには、ミサイルと地形との間で当たり判定処理を行い、垂直方向の速度だけを変化させます（Fig. 4-37）。地形があるときには垂直方向の速度を0にすれば、水平方向の速度だけが残るので、ミサイルは水平に進みます（Fig. 4-37-①）。地形がないときには垂直方向の速度をそのままにすれば、水平方向と垂直方向の速度が合成されて、ミサイルは斜めに落下します（Fig. 4-37-②）。

Fig. 4-35 地形に沿って飛ぶミサイル

自機
地形がないときには
斜めに落下する
地形があるときには地形
に沿って水平に飛ぶ
ミサイル
地上物
地形

Fig. 4-36 水平方向と垂直方向の速度

水平方向の速度
垂直方向の速度
ミサイルの速度は、
水平方向の速度と、
垂直方向の速度とを
合成したもの

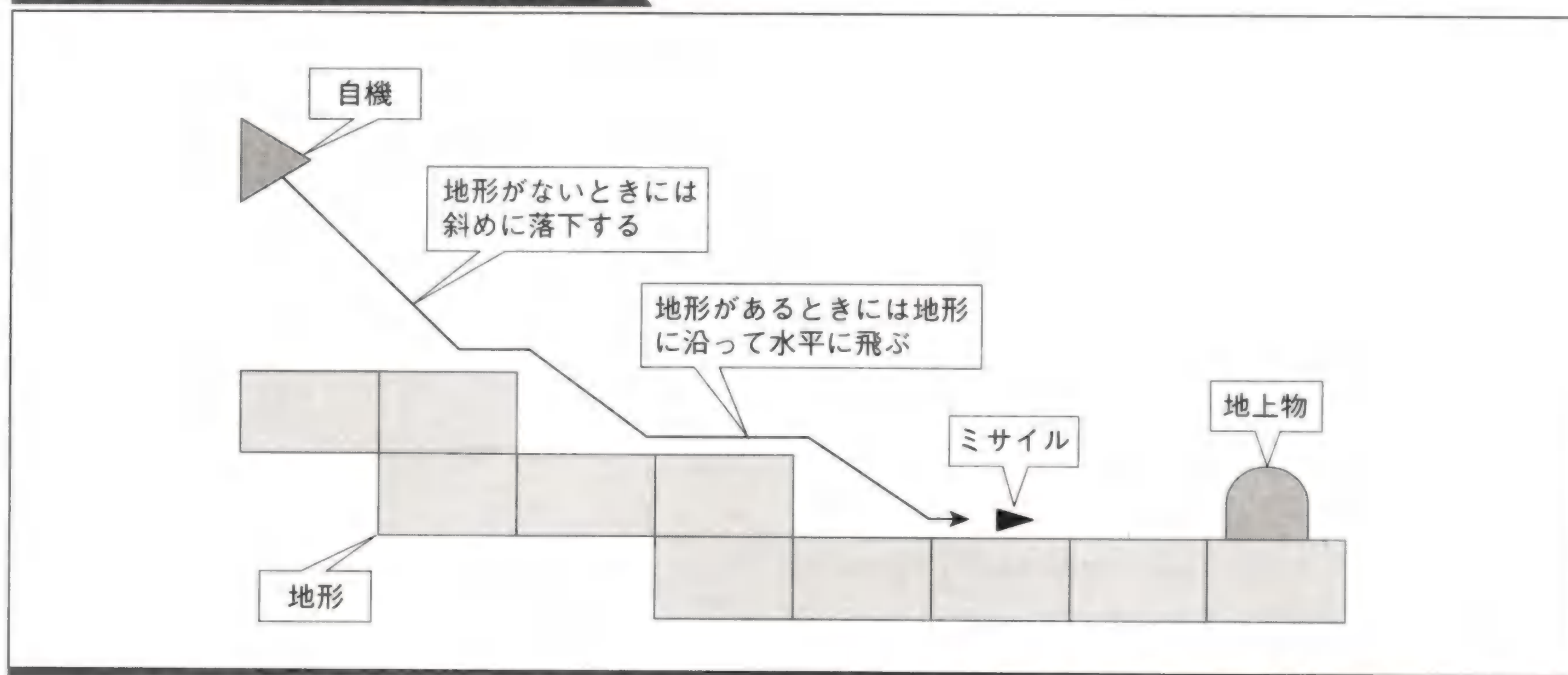
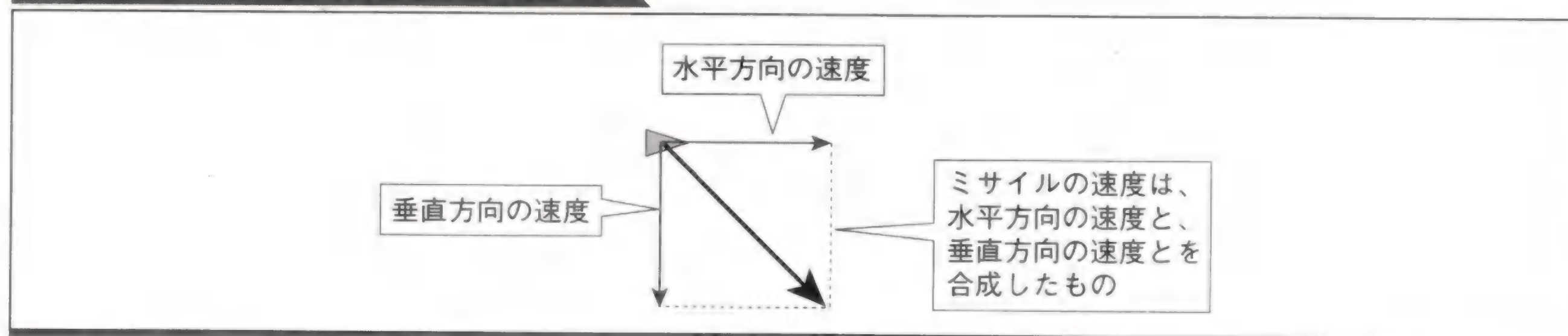



Fig. 4-37 地形の有無に応じた速度の変化

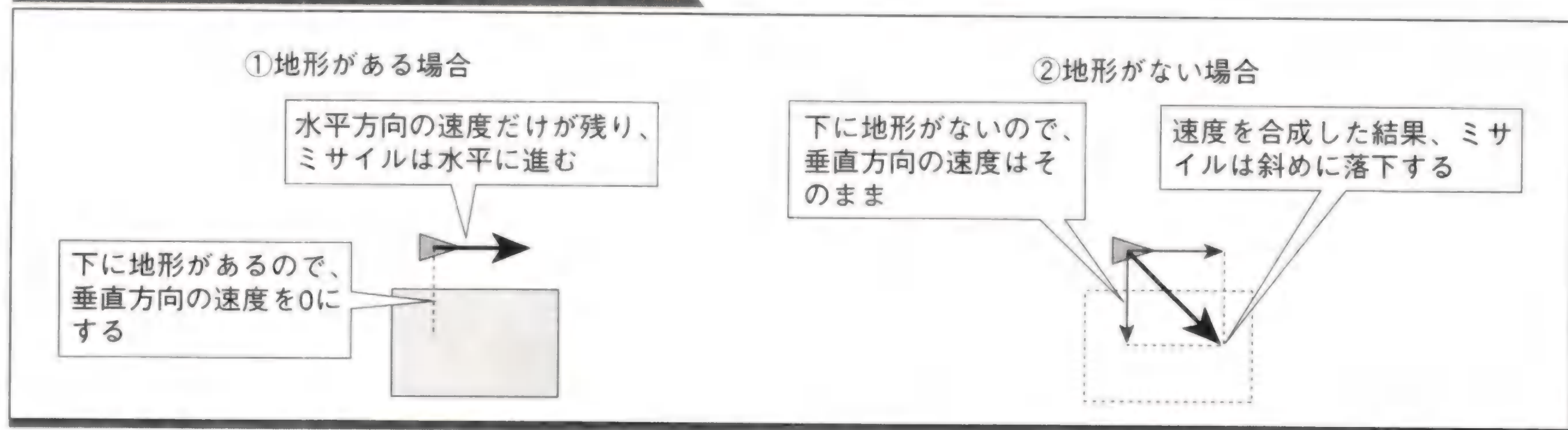



Fig. 4-37は横スクロールゲームの場合ですが、縦スクロールゲームの場合も要領は同じです。縦スクロールゲームでは、地形の有無に応じて水平方向の速度を変化させます。

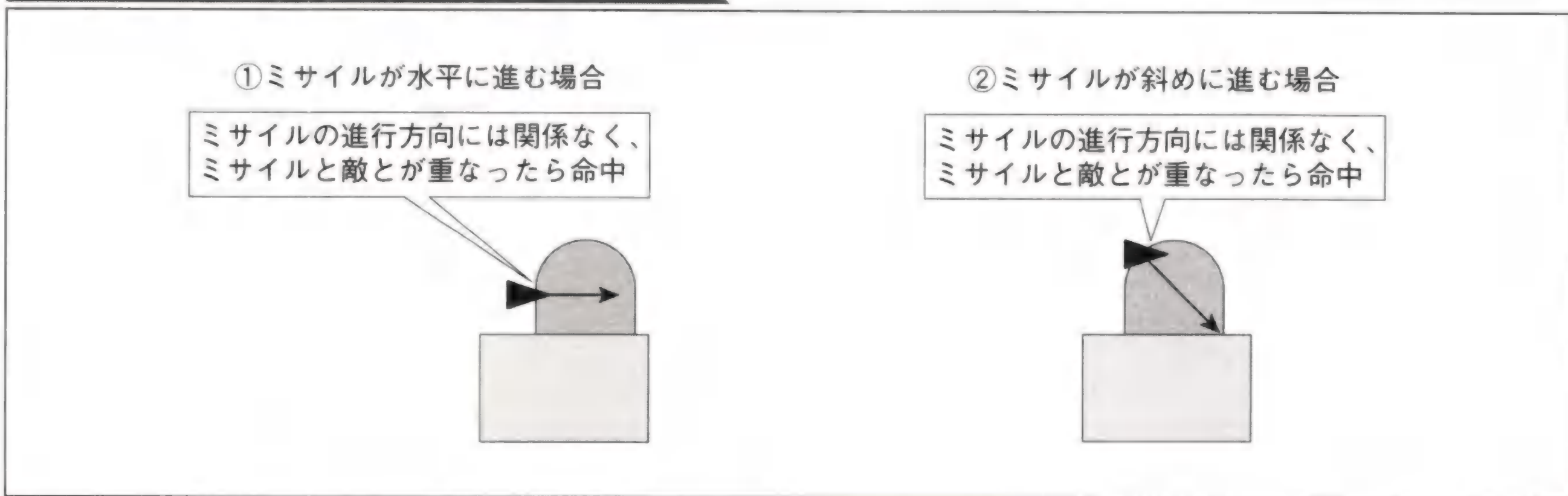
■ ミサイルの当たり判定

ミサイルと敵との当たり判定処理に関しては、地形との当たり判定処理を行って、ミサイルの速度を決定したあとに行います（Fig. 4-38）。ミサイルが水平に進む場合も、斜めに落下する場合も、ミサイルと敵とが重なったら命中したと見なします。

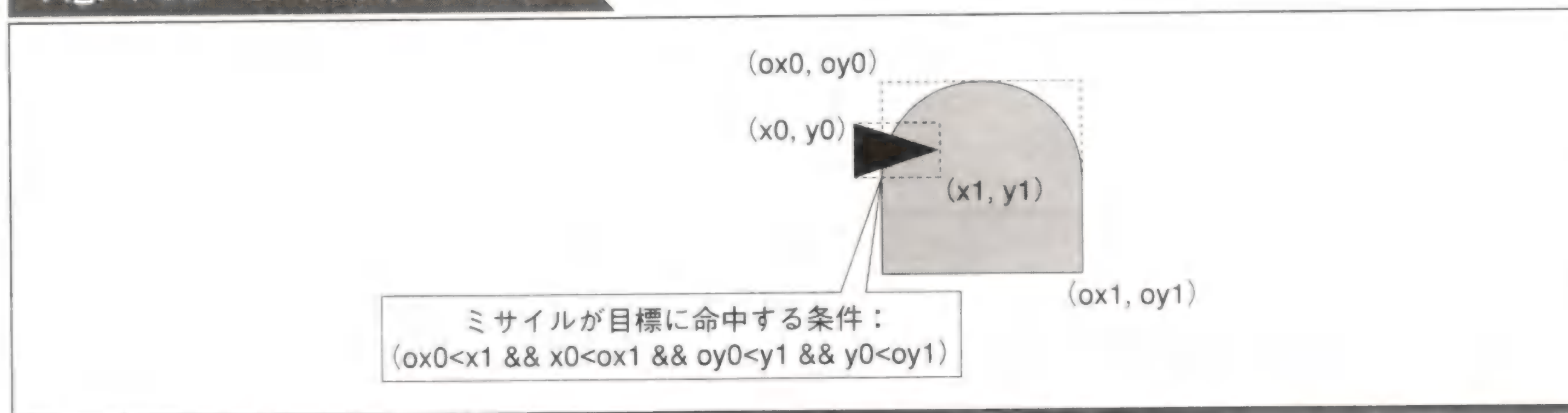
ミサイルと地形、およびミサイルと地形との当たり判定処理は、自機と弾の当たり判定処理などと同じ方法で行えます。たとえば、ミサイルの左上座標を（x0, y0）、右下座標を（x1, y1）、命中対象の左上座標を（ox0, oy0）、右下座標を（ox1, oy1）とすると、ミサイルが対象に命中する条件は次のようになります（Fig. 4-39）。

```
(ox0<x1 && x0<ox1 && oy0<y1 && y0<oy1)
```

List 4-14は地形に沿って飛ぶミサイルの動きをまとめたプログラムです。まず地形との当たり判定処理を行ってミサイルの進行方向を決め、ミサイルを進めます。次に敵との当たり判定処理を行い、敵に当たった場合にはダメージを与えます。

Fig. 4-38 ミサイルと敵との当たり判定処理

サンプル

地形に沿って飛ぶミサイル → P. 317

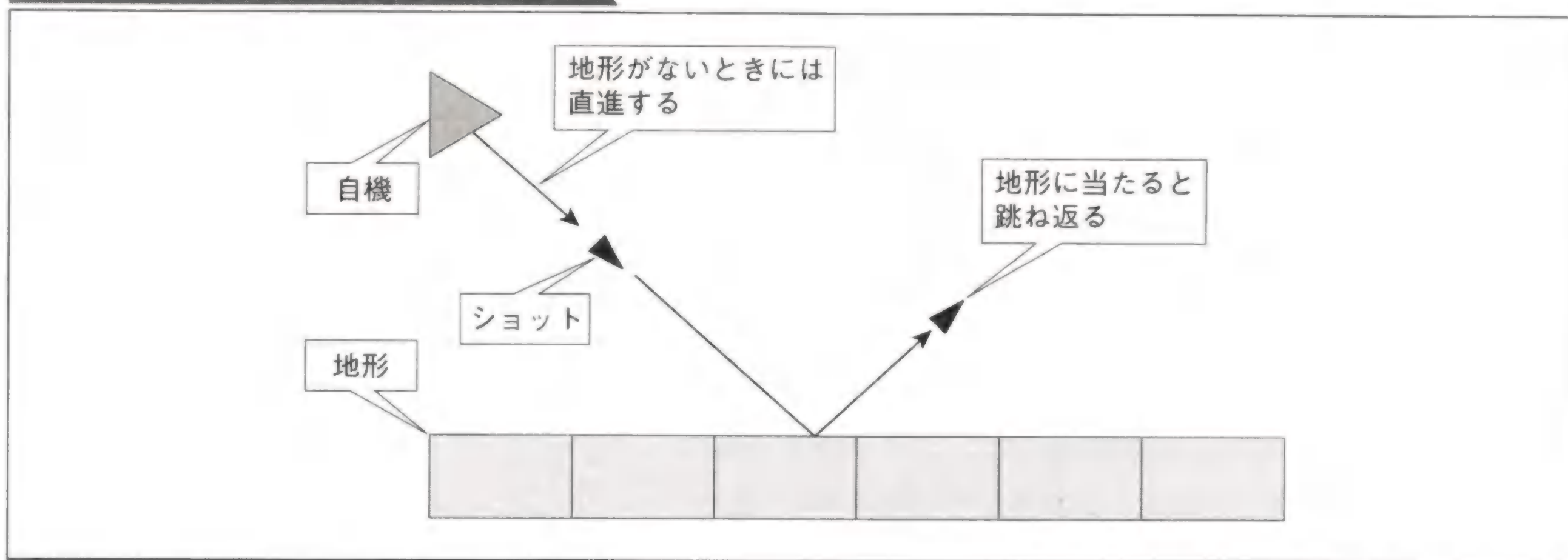
Fig. 4-39 ミサイルが命中する条件

List 4-14 地形に沿って飛ぶミサイル

```
void MoveGroundMissile(
    float& x0, float& y0,       // ミサイルの左上座標
    float& x1, float& y1,       // ミサイルの右下座標
    float& vx, float& vy,       // ミサイルの速度（水平、垂直）
    float gx0[], float gy0[],   // 地形の左上座標
    float gx1[], float gy1[],   // 地形の右下座標
    int num_ground,             // 地形の数
    float ex0[], float ey0[],   // 敵の左上座標
    float ex1[], float ey1[],   // 敵の右下座標
    int num_enemy               // 敵の数
) {
    // 地形との当たり判定処理：
    // ミサイルを垂直方向に進めた座標について、
    // 地形に当たるかどうかを判定する。
    // 地形に当たった場合には、垂直方向の速度を0にして、
    // ループから抜ける。
    for (int i=0; i<num_ground; i++) {
        if (gx0[i]<x1 && x0<gx1[i] &&
            gy0[i]<y1+vy && y0+vy<gy1[i]) {
            vy=0;
            break;
        }
    }

    // ミサイルを進める
    x0+=vx; y0+=vy;
    x1+=vx; y1+=vy;

    // 敵との当たり判定処理：
```

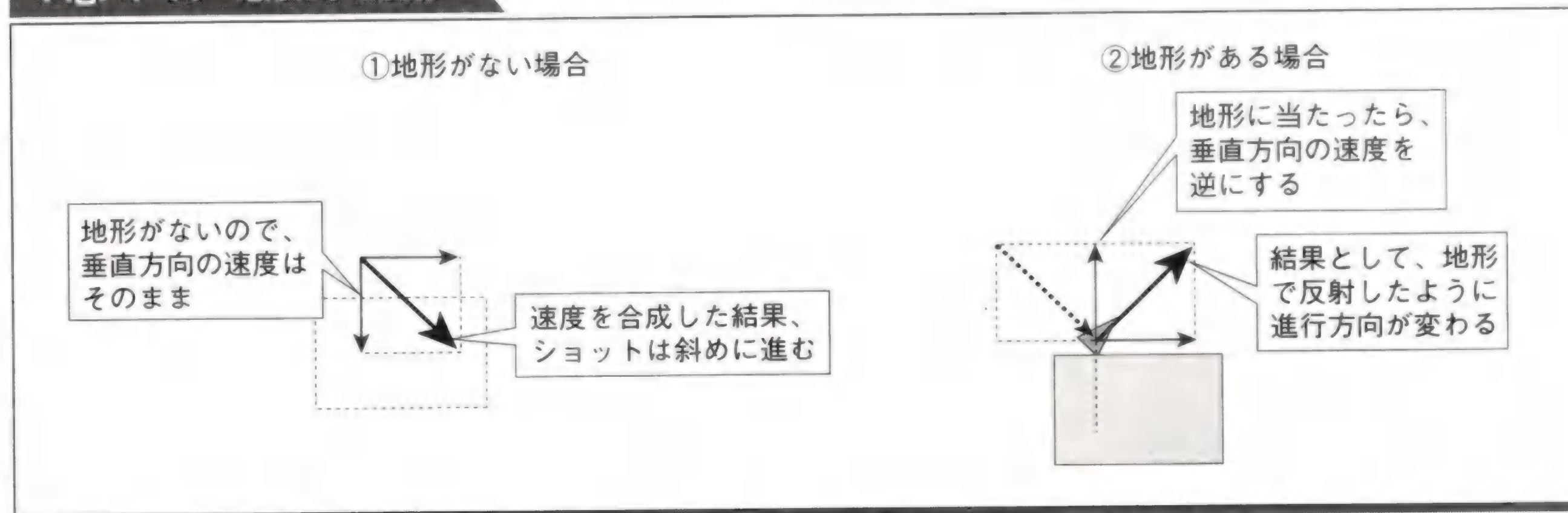
```
        // 敵に当たった場合には、敵にダメージを与えて、
        // ループから抜ける。
        // ダメージの具体的な処理はDamageEnemy関数で行うとする。
        for (int i=0; i<num_enemy; i++) {
            if (ex0[i]<x1 && x0<ex1[i] &&
                ey0[i]<y1 && y0<ey1[i]) {
                DamageEnemy(i);
                break;
            }
        }
    }
```

地形で反射するショット

地形に当たると跳ね返って、進行方向が変わるショットです (Fig. 4-40)。ここでは横スクロールゲームの場合を例に説明しますが、縦スクロールゲームでも「フェリオス (→ P. 332)」などにはこのようなショットがあります。「フェリオス」の場合には左右両方の壁でショットが反射するので、結果としてショットはジグザグに進みます。

Fig. 4-40 地形で反射するショット

地形で反射するショットの考え方は、地形に沿って飛ぶミサイル (→ P. 148) の考え方に似ています。すなわち、ショットの速度を水平方向と垂直方向に分けて考えます (Fig. 4-41)。地形がない場合には、水平方向と垂直方向の速度を合成して、ショットを斜めに進めます (Fig. 4-41-①)。地形がある場合には、垂直方向の速度だけを逆にします (Fig. 4-41-②)。すると、ちょうど地形で反射したように、ショットの進行方向が変わります。

あとの処理は地形に沿って飛ぶミサイルと同じです。地形で反射するショットのプログラムをList 4-15にまとめました。

Fig. 4-41 地形による反射

サンプル

地形で反射するショット　→ P. 317

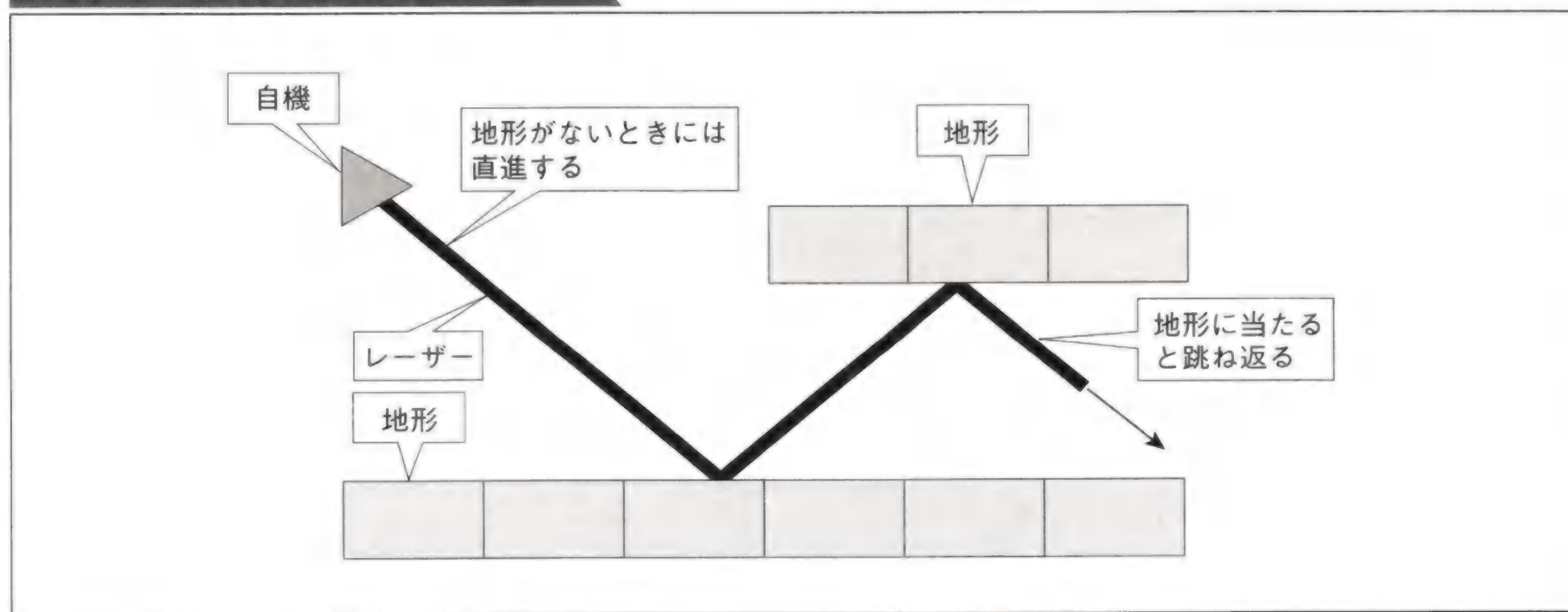
List 4-15 地形で反射するショット

```
void MoveReflectShot(
    float& x0, float& y0,        // ショットの左上座標
    float& x1, float& y1,        // ショットの右下座標
    float vx, float vy,          // ショットの速度(水平、垂直)
    float gx0[], float gy0[],    // 地形の左上座標
    float gx1[], float gy1[],    // 地形の右下座標
    int num_ground,              // 地形の数
    float ex0[], float ey0[],    // 敵の左上座標
    float ex1[], float ey1[],    // 敵の右下座標
    int num_enemy                // 敵の数
) {
    // 地形との当たり判定処理：
    // ショットを垂直方向に進めた座標について、
    // 地形に当たるかどうかを判定する。
    // 地形に当たった場合には、垂直方向の速度を逆にして、
    // ループから抜ける。
    for (int i=0; i<num_ground; i++) {
        if (gx0[i]<x1 && x0<gx1[i] &&
            gy0[i]<y1+vy && y0+vy<gy1[i]) {
            vy=-vy;
            break;
        }
    }
```

```
    // ショットを進める
    x0+=vx; y0+=vy;
    x1+=vx; y1+=vy;

    // 敵との当たり判定処理:
    // 敵に当たった場合には、敵にダメージを与えて、
    // ループから抜ける。
    // ダメージの具体的な処理はDamageEnemy関数で行うとする。
    for (int i=0; i<num_enemy; i++) {
        if (ex0[i]<x1 && x0<ex1[i] &&
            ey0[i]<y1 && y0<ey1[i]) {
            DamageEnemy(i);
            break;
        }
    }
}
```

地形で反射するレーザー

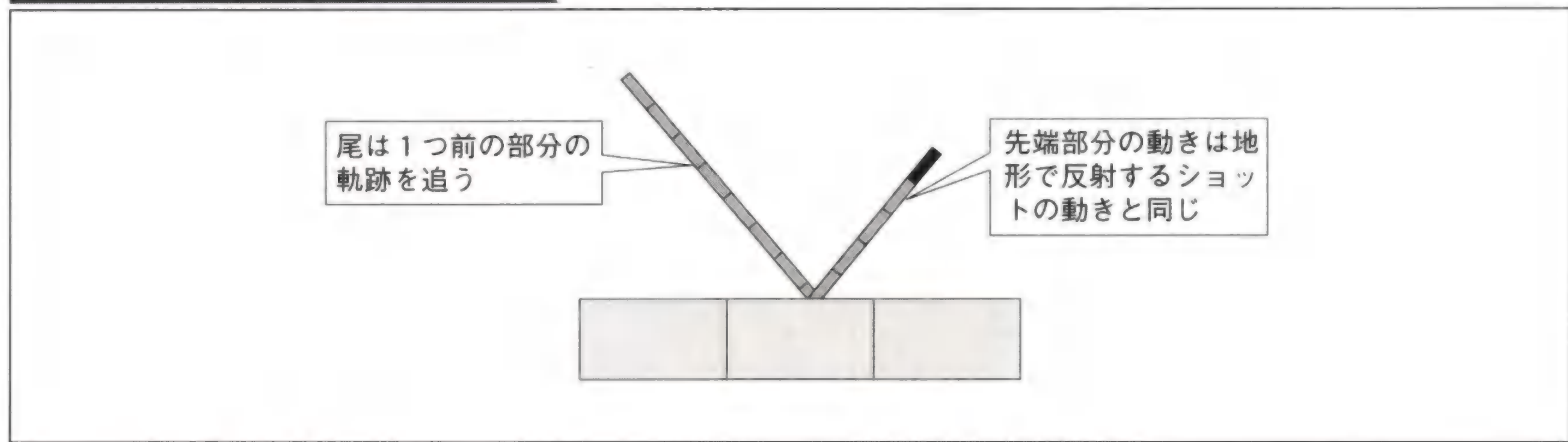
地形に当たると反射するレーザーです(Fig. 4-42)。動きとしては「地形で反射するショット」(→ P. 151)と同じですが、尾があるので見栄えがします。この反射レーザーが出てくる代表的なゲームは「R-TYPE (→ P. 323)」シリーズです。

Fig. 4-42 地形で反射するレーザー

レーザーの先端部分の動きは、地形で発射するショットの動きとまったく同じです（Fig. 4-43）。そこで、あとは「誘導レーザー」（→ P. 46）と同じ方法で尾の部分を動かせば、地形で反射するレーザーが実現できます。尾は1つ前の部分の軌跡を追うようにすればよいのです。

List 4-16は地形で反射するレーザーのプログラムです。

サンプル

地形で反射するレーザー → P. 317

Fig. 4-43 レーザーの描画方法

List 4-16 地形で反射するレーザー

```
// レーザーの各部分を表す構造体
typedef struct LASER_STRUCT {
    float X, Y;                  // 左上座標
    float W, H;                  // 幅と高さ
    float VX, VY;                // 速度
    float OldX, OldY;            // 古い座標
    struct LASER_STRUCT* Prec;   // 1つ前の部分
                                 // （先頭部分の場合にはNULL）
} LASER_TYPE;

// レーザーを動かす関数
void MoveReflectLaser(
    LASER_TYPE* laser,           // レーザーの1つの部分
    float gx0[], float gy0[],    // 地形の左上座標
    float gx1[], float gy1[],    // 地形の右下座標
    int num_ground               // 地形の数
) {
    // 先端部分の場合：
    // 地形で反射するショットの処理と同じ。
    if (!laser->Prec) {

        // 地形との当たり判定処理：
```

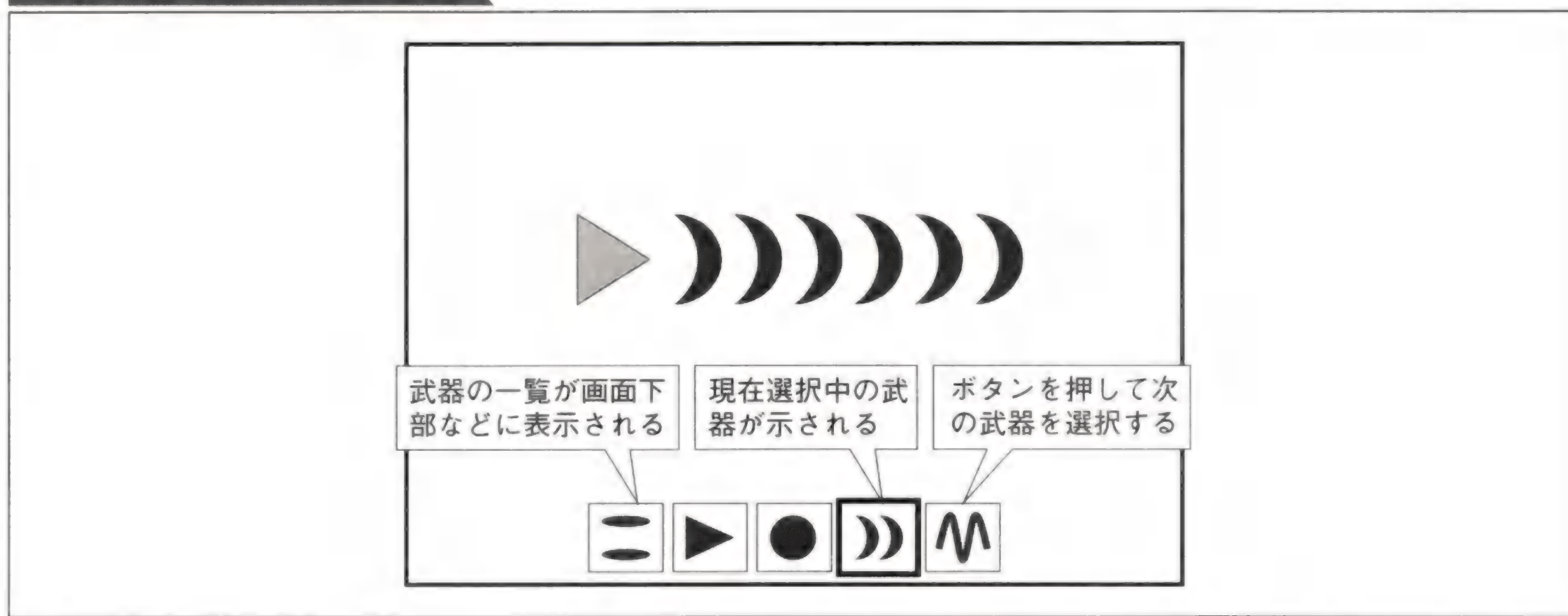
```
        // 先端部分を垂直方向に進めた座標(X, Y+VY)について、
        // 地形に当たるかどうかを判定する。
        // 地形に当たった場合には、垂直方向の速度を逆にして、
        // ループから抜ける。
        for (int i=0; i<num_ground; i++) {
            if (gx0[i]<laser->X+laser->W &&
                laser->X<gx1[i] &&
                gy0[i]<laser->Y+laser->H+laser->VY &&
                laser->Y+laser->VY<gy1[i]
            ) {
                laser->VY=-laser->VY;
                break;
            }
        }

        // 先端部分を進める
        laser->X+=laser->VX;
        laser->Y+=laser->VY;
    }

    // 先端部分以外の場合：
    // 1つ前の部分を追いかける。
    // 1つ前の部分の古い座標へ移動する。
    else {
        laser->X=laser->Prec->OldX;
        laser->Y=laser->Prec->OldY;
    }
}
```

武器の切り替え

攻撃パターンが多く用意されているゲームでは、武器を切り替えながら使うことがあります(Fig. 4-44)。武器を切り替えるゲームでは、たいてい武器の一覧がアイコン表示されていて、持っている武器や現在選択中の武器がわかるようになっています。

多くの場合、武器の切り替えはボタンで行います。ボタンを押すと、一覧のなかで現在選択中の武器の次の武器にフォーカスが移り、その武器が使えるようになります。

List 4-17は武器の切り替え処理をまとめたものです。ボタンを押しっぱなしにしたときに武器が次々と切り替わってしまうのを防ぐため、「基本のショット操作」(→ P. 114)と同じように、前回のボタンの状態を保存しておきます。

Fig. 4-44 武器の切り替え

List 4-17 武器の切り替え

```
// 武器の数
#define MAX_WEAPONS 5

// 武器を切り替える処理
void SelectWeapon(
    bool button  // ボタンの状態（押したときtrue）
) {
    static int weapon_id=0;          // 現在選択中の武器の番号
    static bool prev_button=false;   // 前回のボタンの状態

    // 武器を切り替える：
    // 前回ボタンを押しておらず、今回押していたら、
    // 武器の切り替えを行う。
    if (!prev_button && button) {
        weapon_id=(weapon_id+1)%MAX_WEAPONS;
    }

    // 武器のアイコンを表示する：
    // 全アイコンを表示し、選択中のアイコンは目立たせる。
    // 表示の具体的な処理はDrawIcon関数とFocusIcon関数で
    // 行うとする。
    for (int i=0; i<MAX_WEAPONS; i++) DrawIcon(i);
    FocusIcon(weapon_id);

    // 今回のボタンの状態を保存する
    prev_button=button;
}
```

方向切り替えによる全方位射撃

ボタンを押している間は自機が旋回し、攻撃方向を360度の範囲で自由に変えることができるという、ユニークな攻撃方法です(Fig. 4-45)。

この攻撃方法は「ゼロガンナー(→ P. 329)」シリーズに見られます。「ゼロガンナー」シリーズは横画面、見下ろし型3D表示のシューティングゲームです。この全方位射撃があるおかげで、横画面の狭さを感じさせることなく、画面全体を広々と使って戦えるゲームにしあがっています。

「ゼロガンナー」シリーズの場合は「ターンマーカー」という一種の照準を使って、旋回の中心を表示します(Fig. 4-46)。旋回ボタンを押すと、自機の前方にターンマーカーが出現します。この状態で自機を移動すると、自機はターンマーカーのほうを向くように旋回します(Fig. 4-47)。ターンマーカー自体は移動しませんが、自機の向きに合わせて回転します。

Fig. 4-45 方向切り替えによる全方位射撃

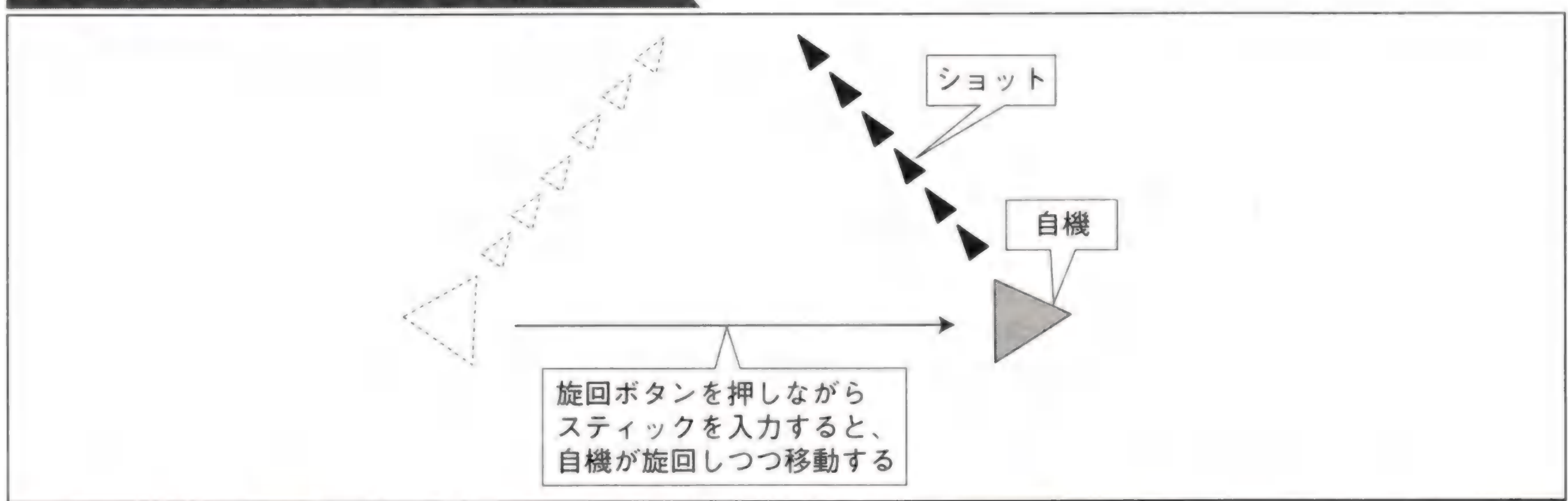



Fig. 4-46 ターンマーカー

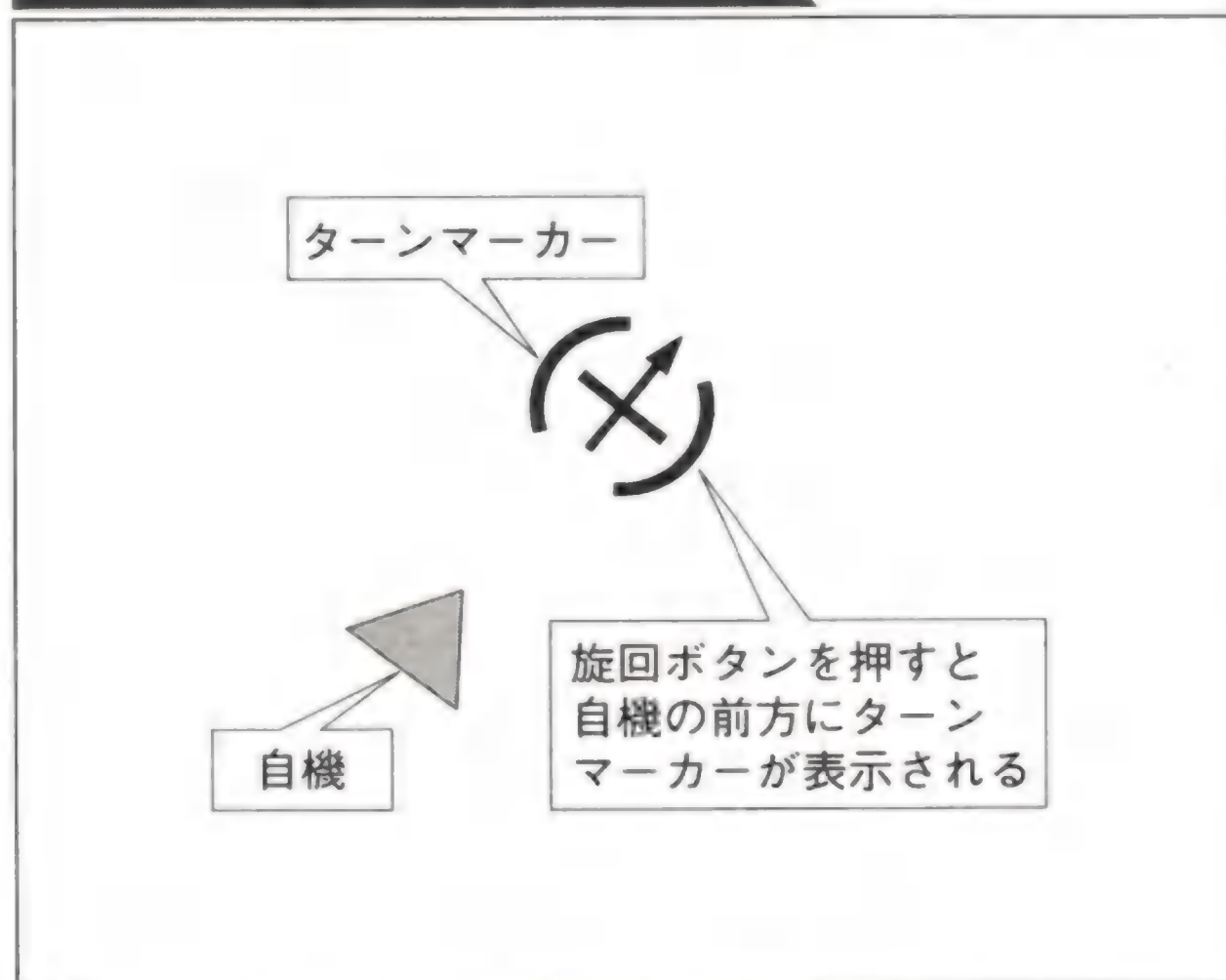
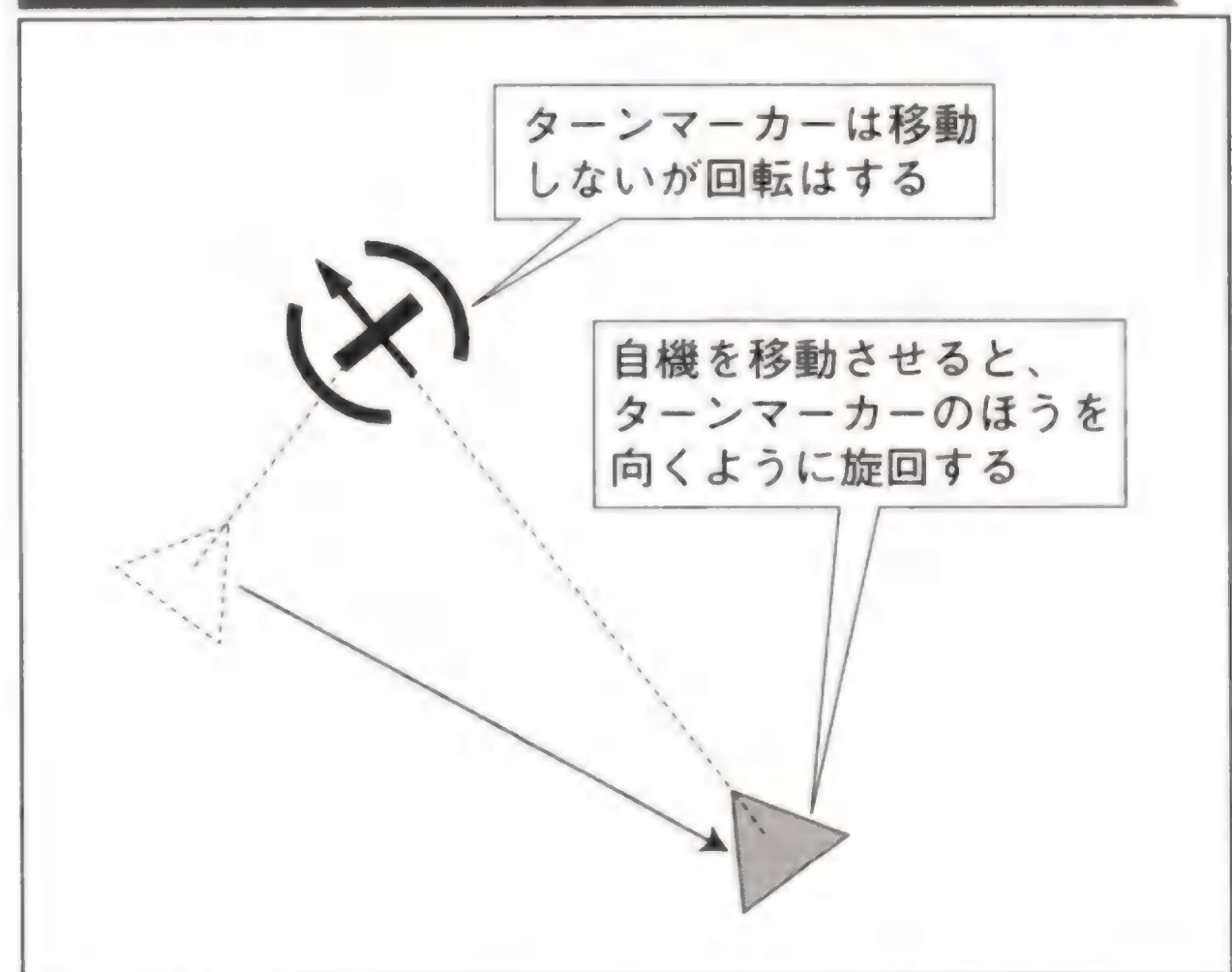



Fig. 4-47 ターンマーカーを中心とした旋回

自機が移動したときの旋回角度は、Fig. 4-48のように求めることができます。ターンマーカーの座標を (mx, my)、自機の座標を (x, y) とします。このとき自機とx軸がなす角度rad (ラジアン) は次のように求められます。

rad=atan2 (y-my, x-mx)

atan2はアークタンジェントを求める (タンジェント値からラジアン値への変換を行う) 関数です。

旋回ボタンを押していないときには、自機は以前の旋回角度を保ったまま上下左右に移動します。ここで旋回ボタンを押したときには、自機の座標から逆にターンマーカーの座標を計算して、ターンマーカーを出現させる必要があります。自機とターンマーカーとの距離をdとすると、ターンマーカーの座標 (mx, my) は次のように求められます (Fig. 4-49)。

mx=x-d*cos (rad)
my=y-d*sin (rad)

Fig. 4-48 旋回角度の計算

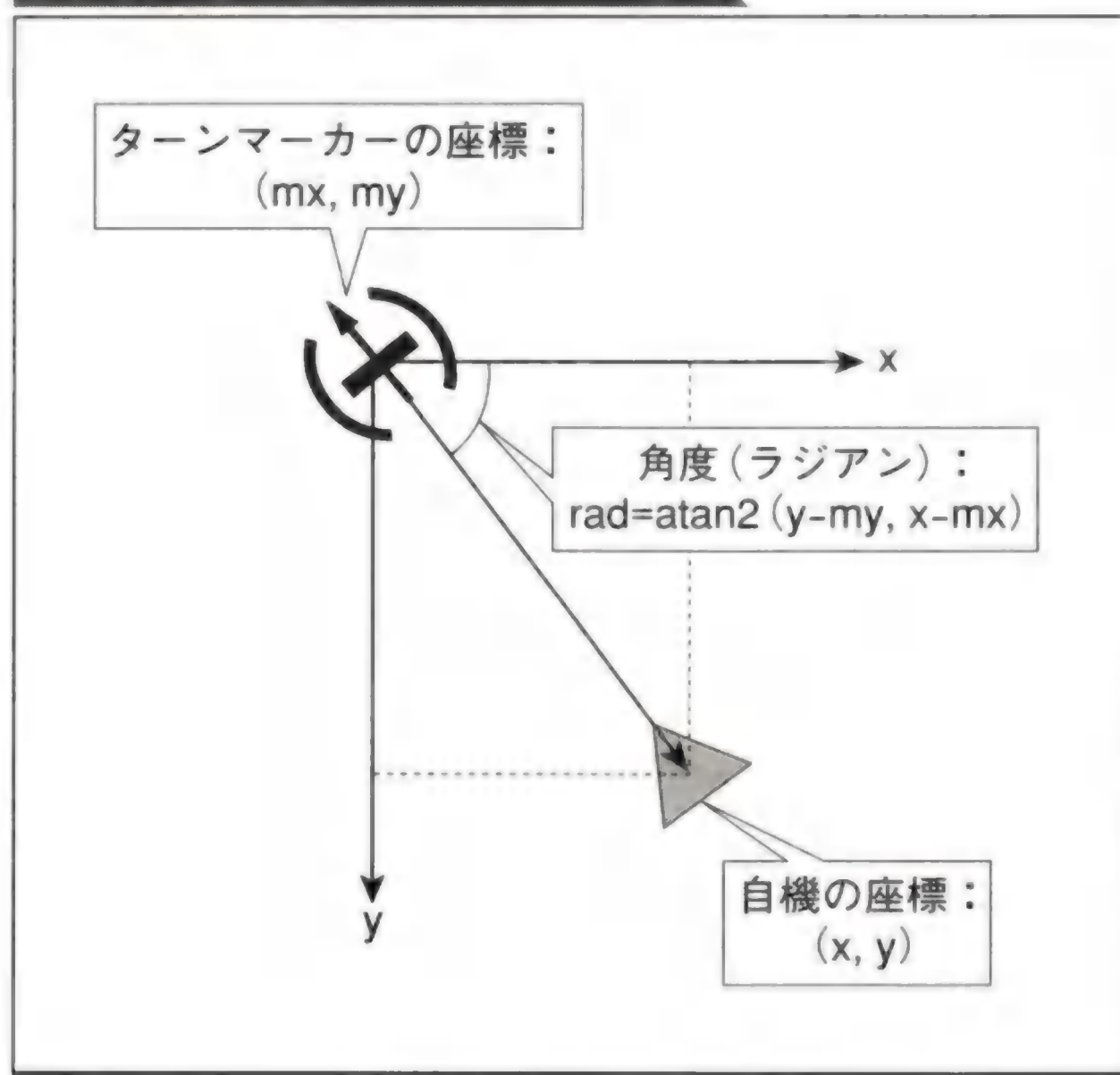
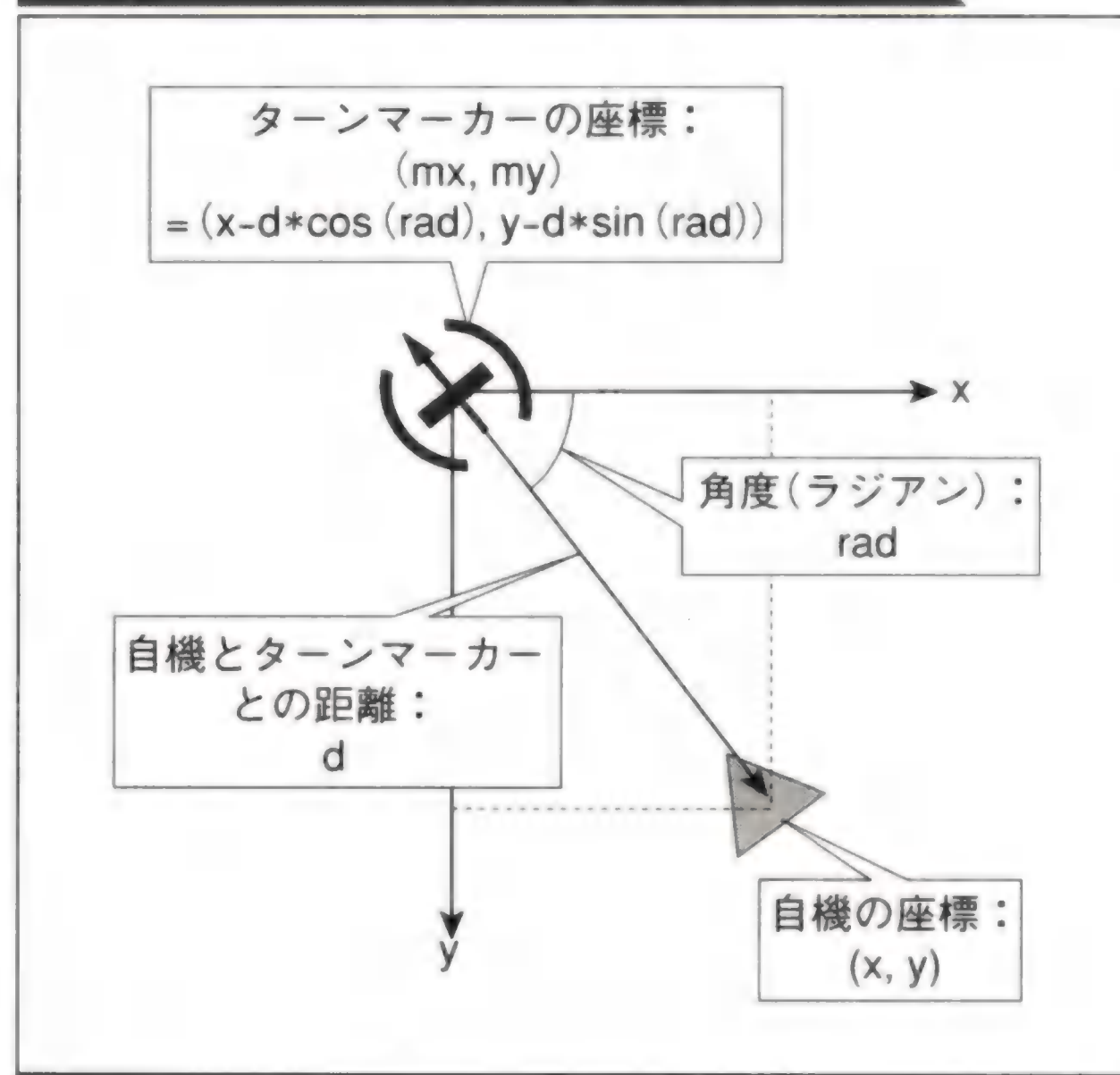



Fig. 4-49 ターンマーカーの座標計算

少し計算が複雑でしたが、方向切り替えによる全方位射撃のプログラムをList 4-18にまとめました。ショットを撃つ方向は、x軸と自機がなす角度とは反対の方向 (180度＝πラジアンずれた方向) になります。

サンプル

方向切り替えによる全方位射撃 → P. 317

List 4-18 方向切り替えによる全方位射撃

```
#include <math.h>

void TurnMarker(
    float& x, float& y,      // 自機の座標
    float speed,             // 自機の速さ
    bool left, bool right,   // スティックの状態(左右方向)
    bool up, bool down,      // スティックの状態(上下方向)
    bool turn_button,        // 旋回ボタンの状態
    bool shot_button         // ショットボタンの状態
) {
    static bool turning;           // 旋回中ならばtrue
    static float mx, my;           // ターンマーカーの座標
    static float rad=M_PI*3/2;     // 旋回角度(M_PIは円周率)
    static float d=10;             // 自機とターンマーカーの距離

    // 旋回の開始:
    // ターンマーカーの座標を算出する。
    if (!turning && turn_button) {
        turning=true;
        mx=x-d*cos(rad);
        my=y-d*sin(rad);
    }

    // 旋回の終了
    if (!turn_button) turning=false;

    // 自機の移動
    if (up   ) y-=speed;
    if (down ) y+=speed;
    if (left ) x-=speed;
    if (right) x+=speed;

    // 旋回中ならば旋回角度を計算する
    if (turning) rad=atan2(y-my, x-mx);

    // ショットの発射:
    // 自機が向いている方向にショットを撃つ。
    // 発射方向はradとは反対方向 (180°=πラジアンずれる)。
    // 発射の具体的な処理はDirectedShot関数で行うとする。
    if (shot_button) DirectedShot(x, y, rad+M_PI);
}
```

Stage 4のまとめ ▶▶

　ショットを中心とした武器について解説しました。基本的には敵が撃つ弾も自機が撃つショットも同じものなので、敵が撃つ弾のアルゴリズムはそのまま自機が撃つショットに応用することが可能です。たとえば誘導弾のアルゴリズム（→ P. 39）を応用すれば、自機に誘導ショットを撃たせることができます。

・ショットは常に撃ち続けるものなので、「撃ったときに何かキモチイイ」ことがあると楽しいゲームになります。たとえばショットの音がいいとか、エフェクトがカッコイイとなれば、プレイヤーもやる気が出るはずです。振動コントローラが使えるなら、強力なショット（レーザーなど）を撃ったときにはコントローラを振動させることもできます。あまり振動させすぎると手首に負担がかかるゲームになってしまいますが、「撃ってる！　撃ってる！」という爽快感をプレイヤーに与えるのは大事なことです。

　というわけで、「ショットはキモチよく、カッコよく！」というのが本章のまとめです。

Stage 05 ▶▶

特殊攻撃
Special Attack

最近のシューティングゲームでは、自機の武器がショットだけということは珍しく、たいていは、あと1つか2つは特殊な攻撃方法が用意されています。本章ではこのようなショット以外の特殊攻撃について解説します。

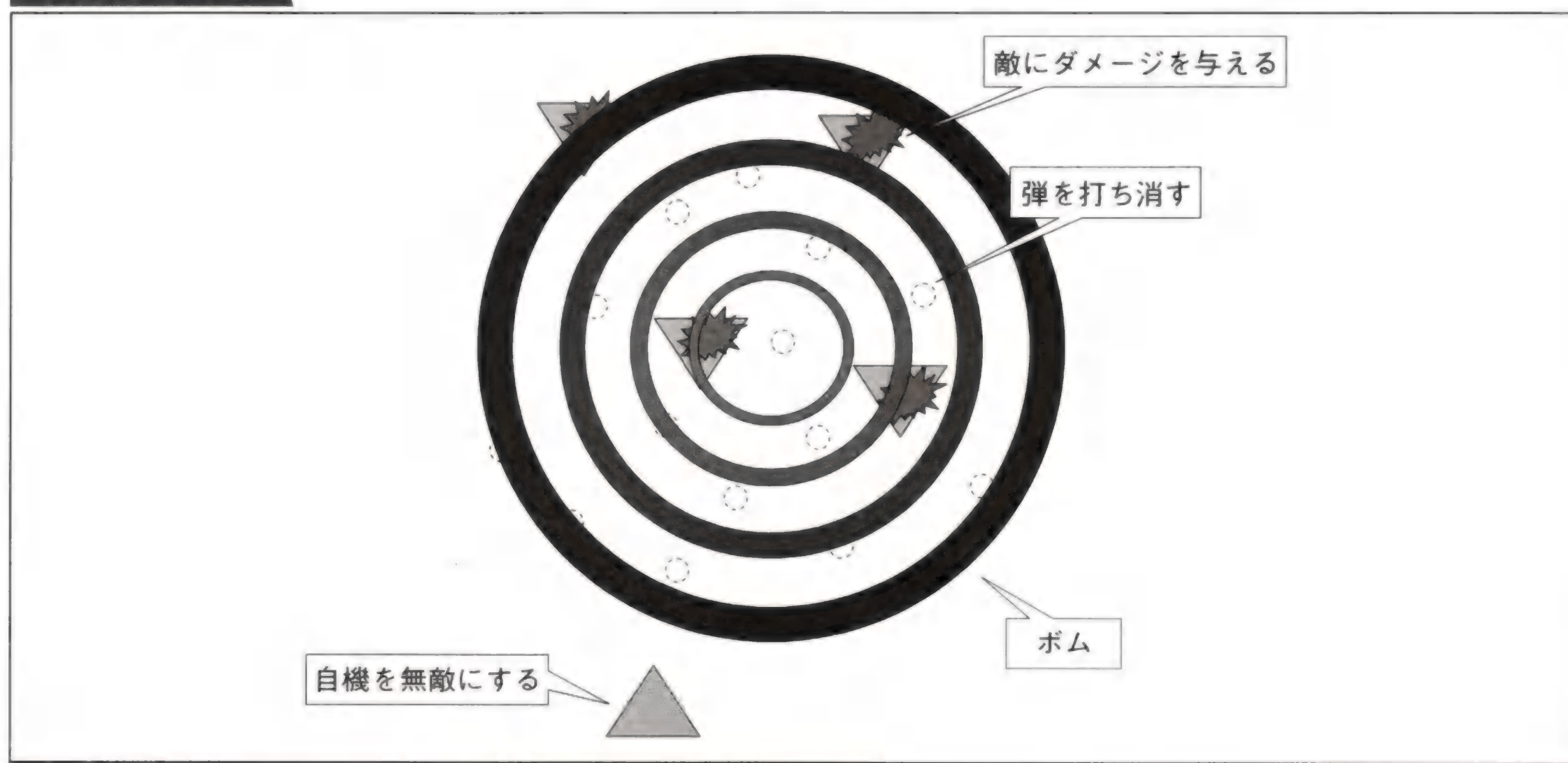
もっともポピュラーな特殊攻撃は「ボム」です。ゲームによってボムの特性には細かい違いがありますが、基本的には広範囲に攻撃判定を持ち、弾を消したり敵に多大なダメージを与えたりといった効果があります。

ボム以外の特殊攻撃は、ゲームによってまさに千差万別です。アイテムを使った特殊な攻撃や、敵や味方をつかんで投げつける攻撃、あるいは敵のレーザーを自機のレーザーで押し返すといった攻撃もあります。

本章ではボムを手始めに、こうしたさまざまな特殊攻撃について解説します。

ボム

ボムは多くのゲームで採用されている特殊攻撃です。もっとも一般的な方式では、ボタンを押すとボムが出現して、画面内の弾を打ち消し、敵に大きなダメージを与えてくれます（Fig. 5-1）。多くのゲームでは、ボムが出ている間は自機が無敵になります。

Fig. 5-1 ボム

ボムの性能はゲームによって微妙に違います。一瞬で出るボムもあれば、少し飛んでから爆発するボムもあります。整理すると、ボムには次のような属性があります。

◇出現までの時間
- 一瞬で出るもの
- 少し飛行してから出るもの

◇出現位置と有効範囲
- 画面中央に出現し、画面全体に有効なもの
- 自機の位置によって出現位置が変わり、一定範囲内に有効なもの

◇有効時間
- 固定時間のもの
- 溜め時間に依存するもの
- ボタンで有効時間が調整できるもの

◇効果
- 弾を消すもの
- 敵にダメージを与えるもの
- 弾や敵をアイテムに変えるもの
- 自機を無敵にするもの

多くのボムはこれらの属性を組み合わせたものです。選択した自機によってボムの性能が変わるゲームもあります。

ボムの処理はゲームによって変わりますが、List 5-1では弾を消して敵にダメージを与える一般的なボムの処理をまとめてみました。

サンプル

ボム →P. 318

List 5-1 ボム

```
void Bomb(
    int& bomb_time,                  // ボムの有効時間
    float x0, float y0,              // ボムの有効範囲の左上座標
    float x1, float y1,              // ボムの有効範囲の右下座標
    float bx0[], float by0[],        // 弾の左上座標
    float bx1[], float by1[],        // 弾の右下座標
    int num_bullet,                  // 弾の数
    float ex0[], float ey0[],        // 敵の左上座標
    float ex1[], float ey1[],        // 敵の右下座標
    int num_enemy                    // 敵の数
) {
    // ボムが有効な場合の処理
    if (bomb_time>0) {

        // 弾を消す：
        // 消去の具体的な処理は、
        // DeleteBullet関数で行うとする。
        for (int i=0; i<num_bullet; i++) {
            if (bx0[i]<x1 && x0<bx1[i] &&
                by0[i]<y1 && y0<by1[i]) {
                DeleteBullet(i);
            }
        }

        // 敵にダメージを与える：
        // ダメージ付与の具体的な処理は、
        // DamageEnemy関数で行うとする。
        for (int i=0; i<num_enemy; i++) {
```

```
            if (ex0[i]<x1 && x0<ex1[i] &&
                ey0[i]<y1 && y0<ey1[i]) {
                DamageEnemy(i);
            }
        }

        // ボムの有効時間を減らす
        bomb_time--;
    }
}
```

ボムとゲームバランス

ボムに関してだけいえば、世の中のシューティングゲームは「ボムがあるゲーム」と「ボムがないゲーム」という2つに分けられます。これら2つのゲームでは、ゲームバランスの調整方法がだいぶ変わってきます。

ボムは弾や敵を消してくれるので、非常に強力な危機回避手段にもなります。そのためボムがあるゲームは、多少きつめのゲームバランスにしておいても、いざとなればボムさえ使えばクリアすることができます。逆にボムがないゲームでは、「ボムがなくては絶対に抜けられない場面」を作らないように注意しなければなりません。

ボムがあるゲームには次の2つの設計方針があります。

①ボムを使わずに全場面を抜けられるようにする
②ボムを使わないと抜けられない場面を含む

おそらくプレイヤーの心理としては、②よりも①のほうを好ましく思うでしょう。①はつまり、このゲームはまったくボムを使わずにクリアできる（少なくとも非常に上手なプレイヤーならば、その可能性がある）ということです。ボムというのはやはり緊急の危機回避手段という印象が強いので、「ボムをまったく使わないでクリアするのが究極のスタイルだ」と考えるプレイヤーは少なくないでしょう。

「では、具体的にどうやってゲームバランスを調整したらいいのか？」という問いに答えるのは非常に難しいのですが、ここではボムの有無と自機の復活方法別に、ゲームバランスの調整に関する注意点をまとめてみました（Fig. 5-2）。けっきょくは何度もプレイしたり、さまざまな技量の人にプレイしてもらったりして調整するしかありません。

Fig. 5-2 ボムの有無と復活方法別のゲームバランス調整法

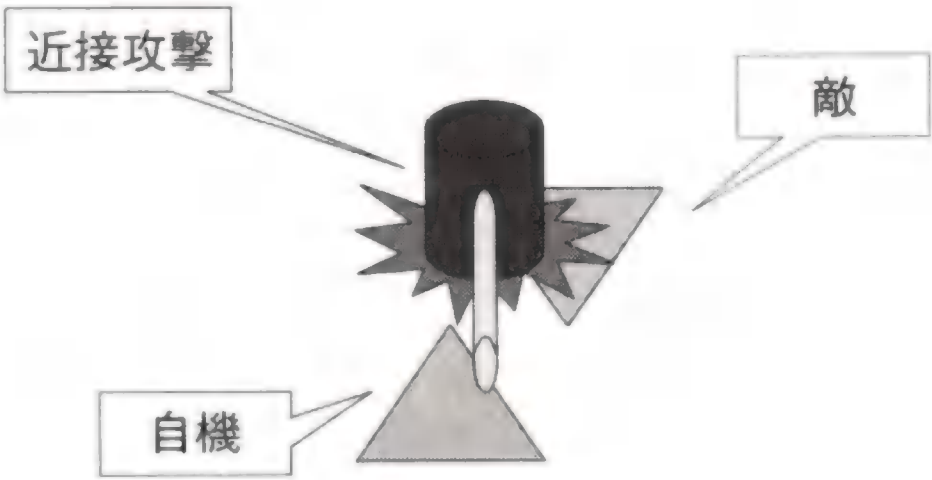
	ボムあり	ボムなし
その場復活	ボムを使うか残機を消費するかすればクリアはできる。 できればボムなし、残機消費なしでクリアできるようにしたい	残機を消費すればクリアはできる。 できれば残機消費なしでクリアできるようにしたい
戻り復活	ボムを使い、かつボムの数が十分ならば、クリアはできる。 できればボムなしでクリアできるようにしたい	ボムや残機を使わないとクリアできない場面を作ってはならない。 普通にプレイしてクリアできるように作ればよい

なお、「その場復活」というのは「自機が死んだときにその場に新しい自機が出現する方式」のことで、「戻り復活」というのは「自機が死んだときにマップ上の決まった位置まで戻される方式」のことです。

ボムやその場復活があるゲームでは、ボムや残機を消費してのプレイまで考えると、バランス調整が難しくなります。一方、ボムもその場復活もないゲームでは「普通にクリアできるかどうか」だけが問題なので、バランス調整の作業自体はシンプルです。ただし、ゲームを難しくするには相当に腕利きのテストプレイヤーが必要になるでしょう。

近接攻撃

自機からパンチやハンマーなどを出して、近くにいる敵を直接たたく攻撃方法です(Fig. 5-3)。「ツインビーヤッホー(→ P. 331)」や「ガンバード(→ P. 324)」「ガンバード2(→ P. 324)」などのゲームに見られます。

Fig. 5-3 近接攻撃

近接攻撃を出す方法はゲームによって違い、ボタン一発で出るゲームもあれば、溜め攻撃扱いになっているゲームもあります。効果はだいたい似たようなもので、自機のすぐ近くに攻撃が出て、少しの間持続し、決まった時間が経つと引っ込むというものです。攻撃が敵に重なると、通常のショットよりも大きなダメージを与えることができます。近接攻撃はダメージが大きいので、危険を冒して敵に接近する理由になります。そのため「敵の隙をついて接近し、近接攻撃をたたき込んで逃げる」というゲーム性が生まれます。

近接攻撃の処理は、持続時間さえうまく扱えば、特に難しいことはありません。List 5-2に近接攻撃のプログラムをまとめます。

サンプル

パンチ →P. 318

List 5-2 近接攻撃

```
// 近接攻撃に関する処理
void Punch(
    bool button,                // ボタンの状態（押したときtrue）
    float px0, float py0,       // 近接攻撃の左上座標
    float px1, float py1,       // 近接攻撃の右下座標
    float ex0[], float ey0[],   // 敵の左上座標
    float ex1[], float ey1[],   // 敵の右下座標
    int num_enemy               // 敵の数
) {
    static bool punching=false;  // 近接攻撃中かどうか
    static int punch_time;       // 近接攻撃を出している時間

    // 近接攻撃の開始：
    // 近接攻撃中ではなく、ボタンが押されていたら、
    // 近接攻撃を出す。
    if (!punching && button) {
        punching=true;
        punch_time=30;
    }

    // 近接攻撃中の処理
    if (punching) {

        // 敵との当たり判定処理：
        // ダメージを与える具体的な処理は、
        // DamageEnemy関数で行うとする。
        for (int i=0; i<num_enemy; i++) {
            if (ex0[i]<px1 && px0<ex1[i] &&
                ey0[i]<py1 && py0<ey1[i]) {
                DamageEnemy(i);
```

```
            }
        }

        // 近接攻撃の表示：
        // 表示の具体的な処理はDrawPunch関数で行うとする。
        DrawPunch();

        // 近接攻撃の持続と終了：
        // 持続時間を過ぎたら近接攻撃を引っ込める。
        if (punch_time==0) punching=false; else punch_time--;
    }
}
```

誘爆

敵が破壊したときの爆風に、ほかの敵を巻き込んで破壊する攻撃方法です（Fig. 5-4）。「グロブダー（→ P. 326）」ではこの誘爆が攻略上の重要なポイントになっています。もっと古いゲームでは「ミサイルコマンド（→ P. 334）」が有名です。「ミサイルコマンド」では自機（自分の基地）が撃てるミサイルの数が制限されているので、敵のミサイルを爆破したときの誘爆を利用していかに弾幕を張るかがカギになっています（Fig. 5-5）。

誘爆は利用するプレイヤーにとっては奥が深い要素ですが、プログラムを作る側にとっては難しいものではありません（Fig. 5-6）。基本的には、敵が破壊されたときに「敵に対して当た

Fig. 5-4 誘爆

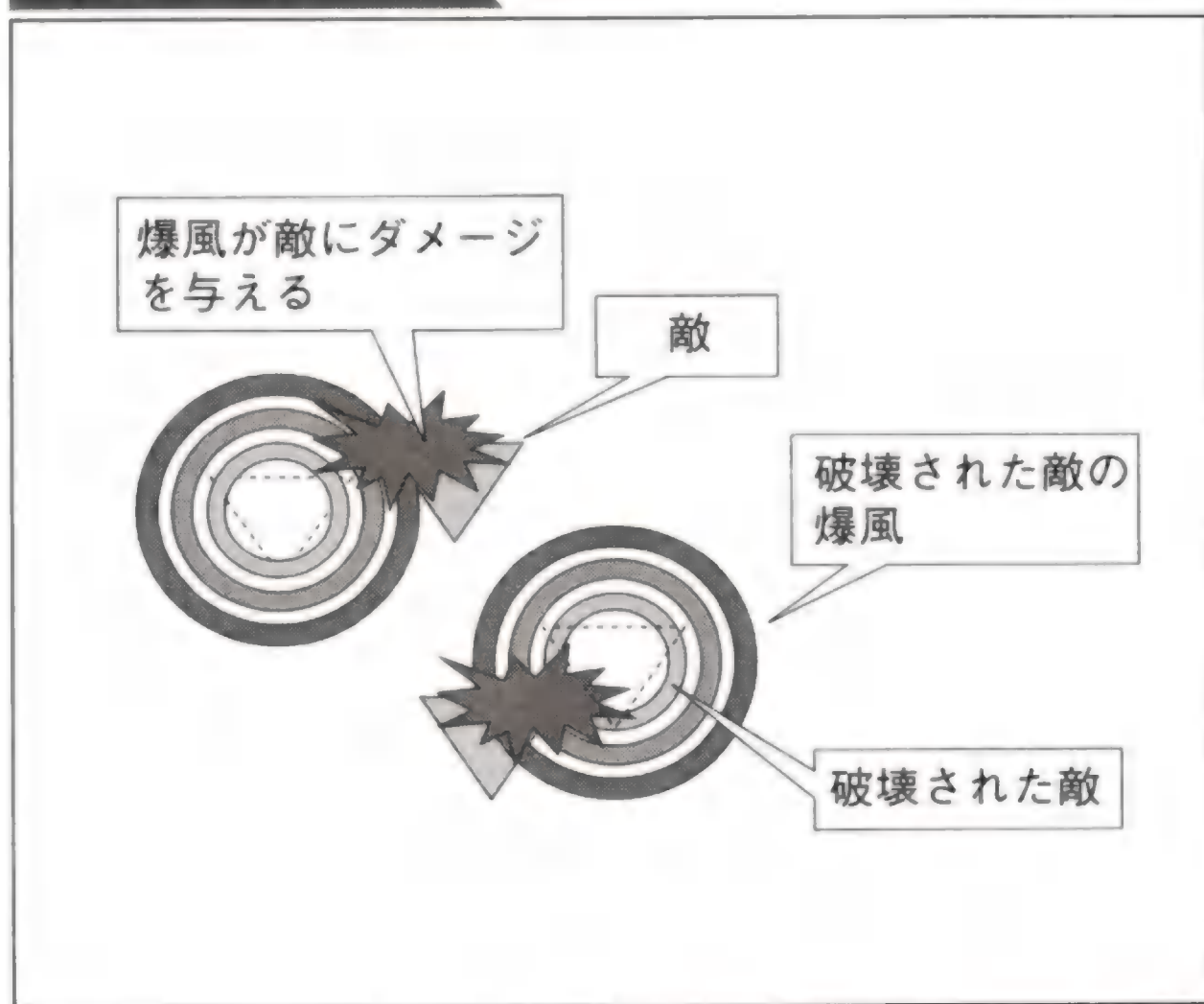
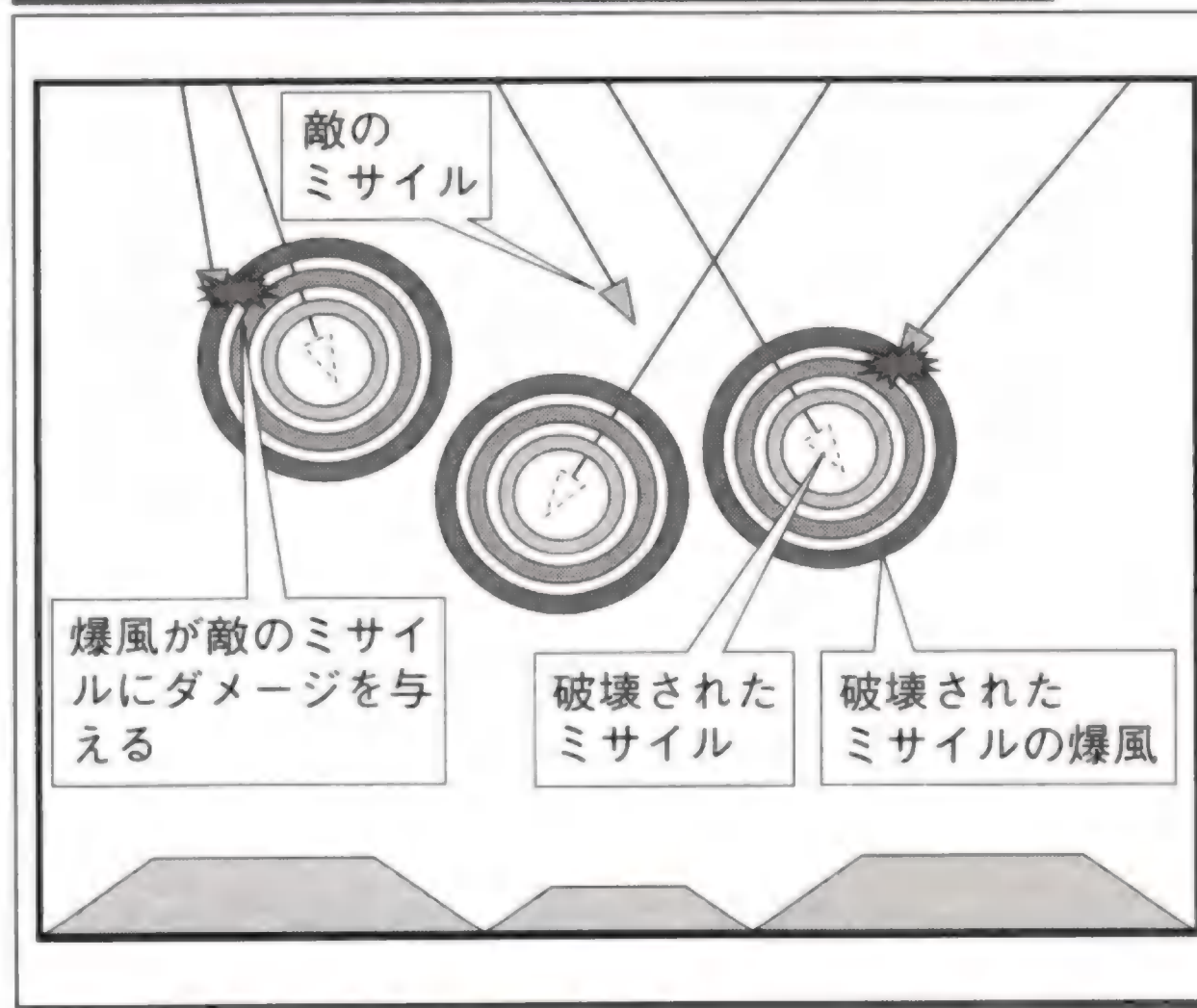
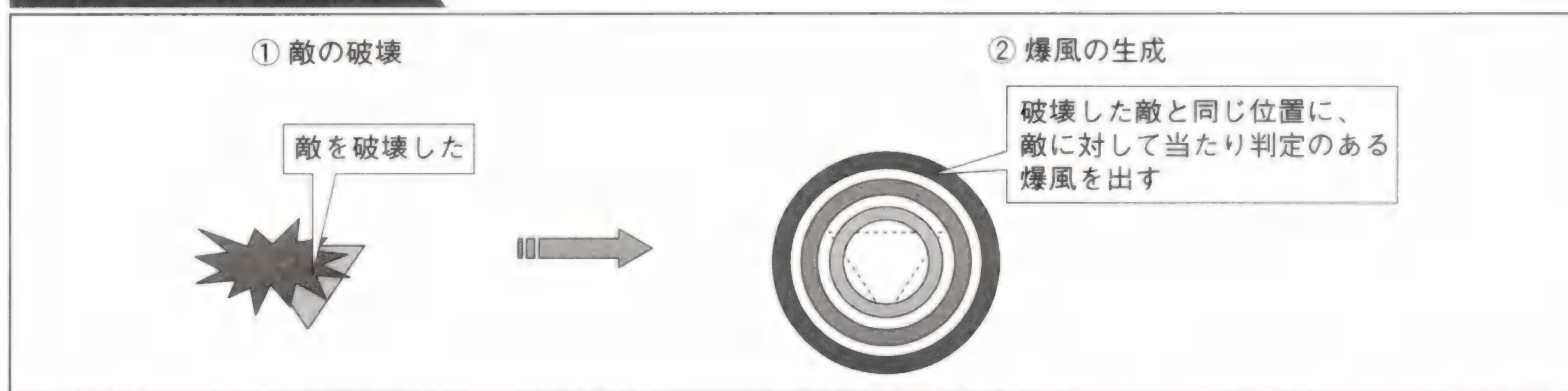



Fig. 5-5 ミサイルコマンドにおける誘爆

り判定のある爆風」を出すだけです。

List 5-3は誘爆の処理をまとめたプログラムです。すべての敵と爆風との間で当たり判定処理を行い、敵が爆風に当たったときには敵を破壊して、その位置に新しい爆風を生成します。

サンプル

誘爆 → P. 318

Fig. 5-6 誘爆の仕組み

List 5-3 誘爆

```
void InducedExplosion(
    float ex0[], float ey0[],  // 敵の左上座標
    float ex1[], float ey1[],  // 敵の右下座標
    int num_enemy,             // 敵の数
    float x0[], float y0[],    // 爆風の左上座標
    float x1[], float y1[],    // 爆風の右下座標
    int num_explosion          // 爆風の数
) {
    // 敵と爆風の当たり判定処理：
    // すべての敵と爆風との間で当たり判定処理を行い、
    // 敵が爆風に当たったら破壊する。
    // そして敵と同じ座標に新たな爆風を生成する。
    // 破壊と生成の具体的な処理は、
    // DestroyEnemy、CreateExplosionの各関数で行うとする。
    for (int i=0; i<num_enemy; i++) {
        for (int j=0; j<num_explosion; j++) {
            if (ex0[i]<x1[j] && x0[j]<ex1[i] &&
                ey0[i]<y1[j] && y0[j]<ey1[i]) {
                DestroyEnemy(i);
                CreateExplosion(ex0[i], ey0[i]);
            }
        }
    }
}
```

アイテムによる特殊攻撃

敵を倒したときに出るアイテムを拾って、何か特殊な攻撃をするというものです。たとえば「パロディウス（→ P. 332）」シリーズでは、アイテム（ベル）を拾うとさまざまな特殊攻撃が発動します。分身が出たりボムが出たりといった普通のものから、メガホンでセリフを叫んで敵を攻撃するといったコミカルなものまであります（Fig. 5-7）。

「パロディウス」のメガホンのような特殊攻撃は、一度発動すると通常のショットが出なくなり、かわりに特殊攻撃が出るようになります。そして一定時間が経つと特殊攻撃は切れて、再び通常のショットに戻ります。このような状態の移り変わり（状態遷移）を図にしたのがFig. 5-8です。現在の状態が通常か特殊かによって攻撃方法を変え、アイテムを拾ったり一定時間が経過したりしたら別の状態へ移行します。

ちなみに、Fig. 5-8のような状態遷移の処理をフローチャート（最近ではあまり使わなくなりましたが）にまとめると、Fig. 5-9のようになります。フローチャートにするとだいぶ複雑になってしまいますが、Fig. 5-8のような2つの状態の間を移行しているだけなので、そう難しい処理ではありません。

Fig. 5-7 アイテムによる特殊攻撃

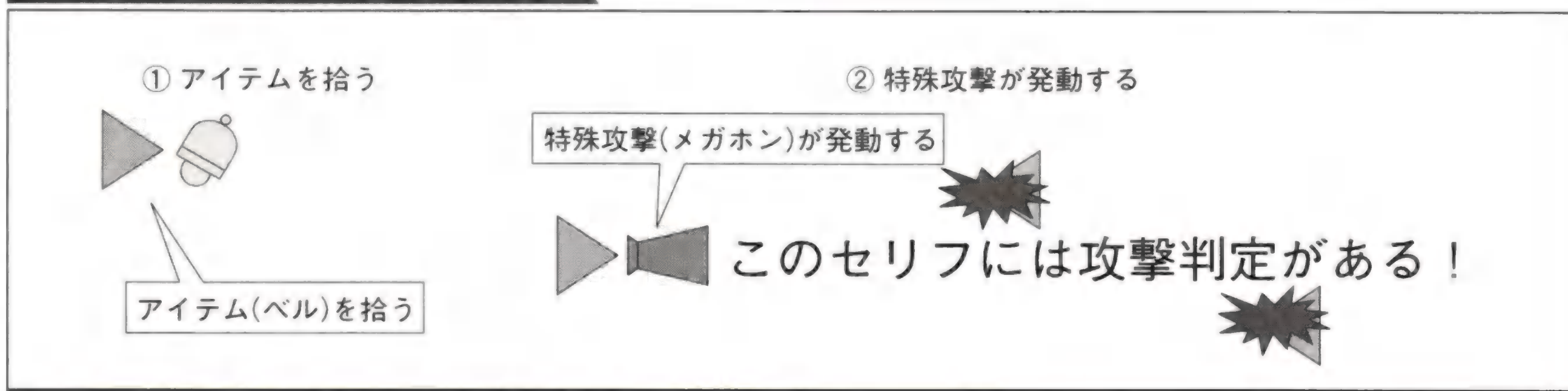



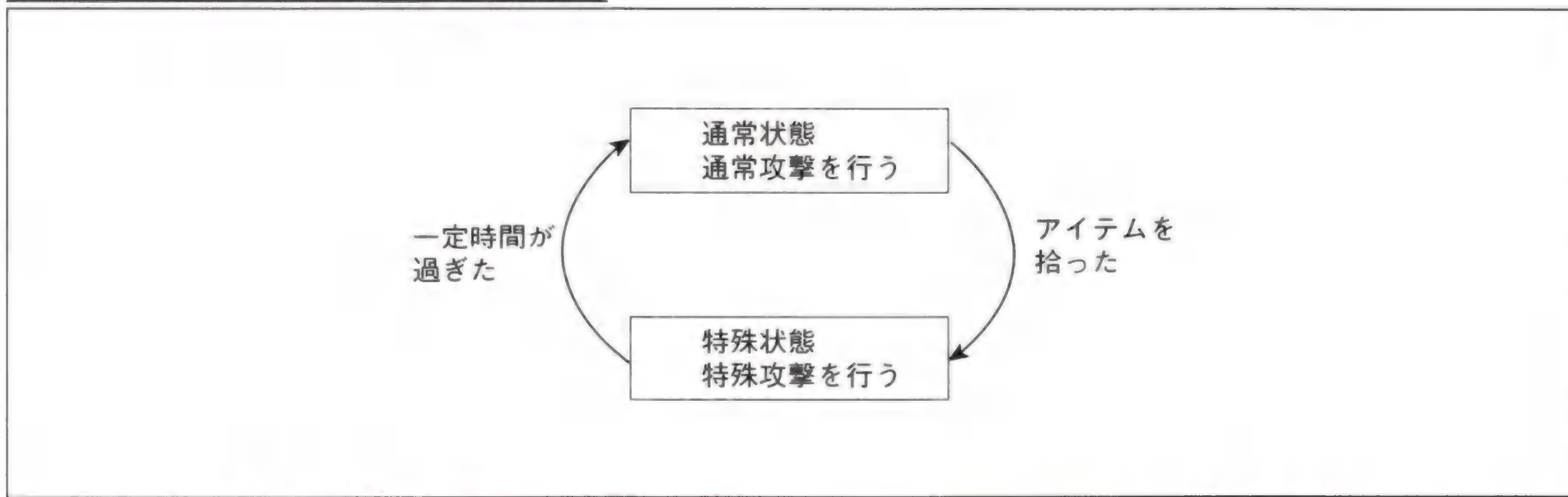
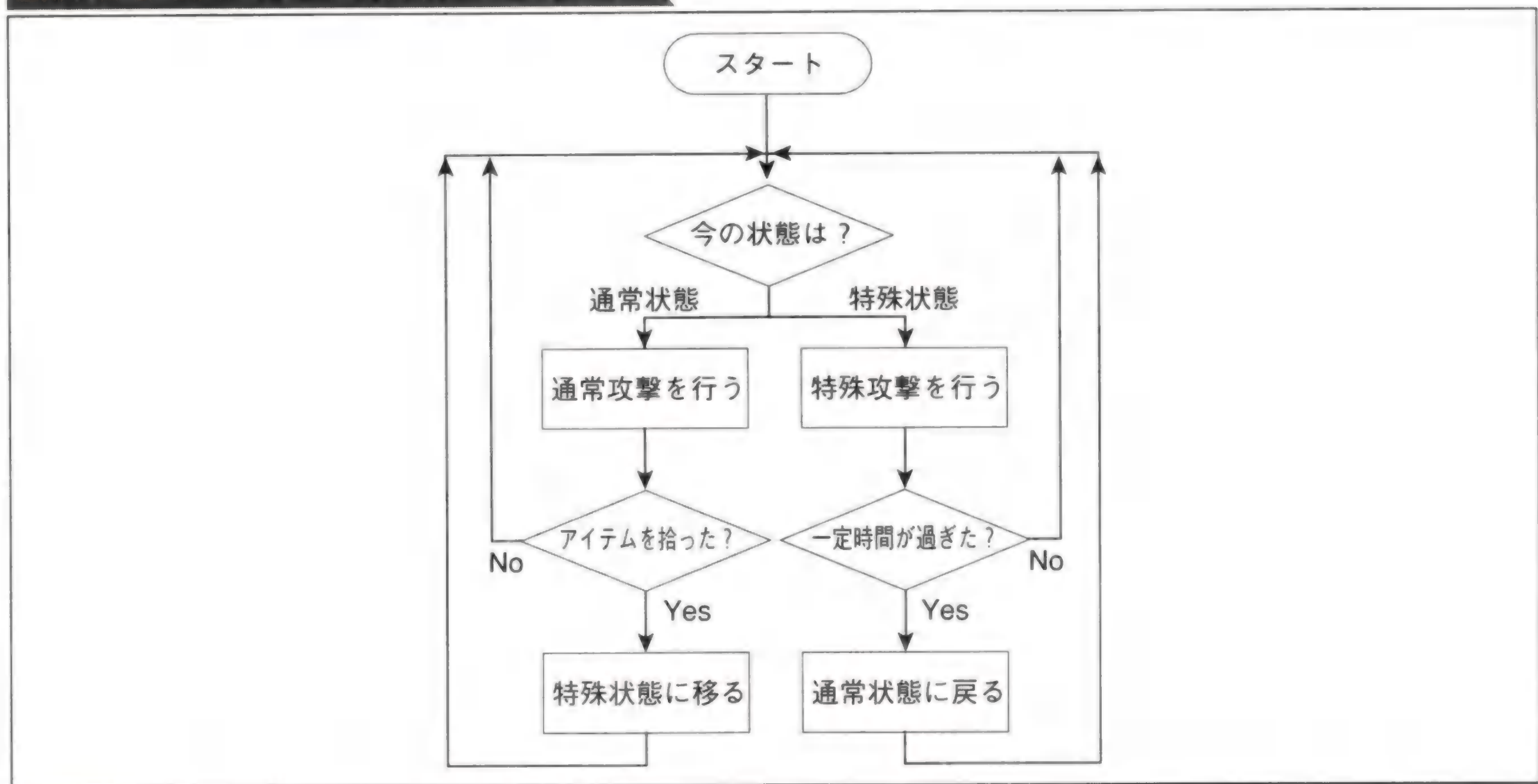
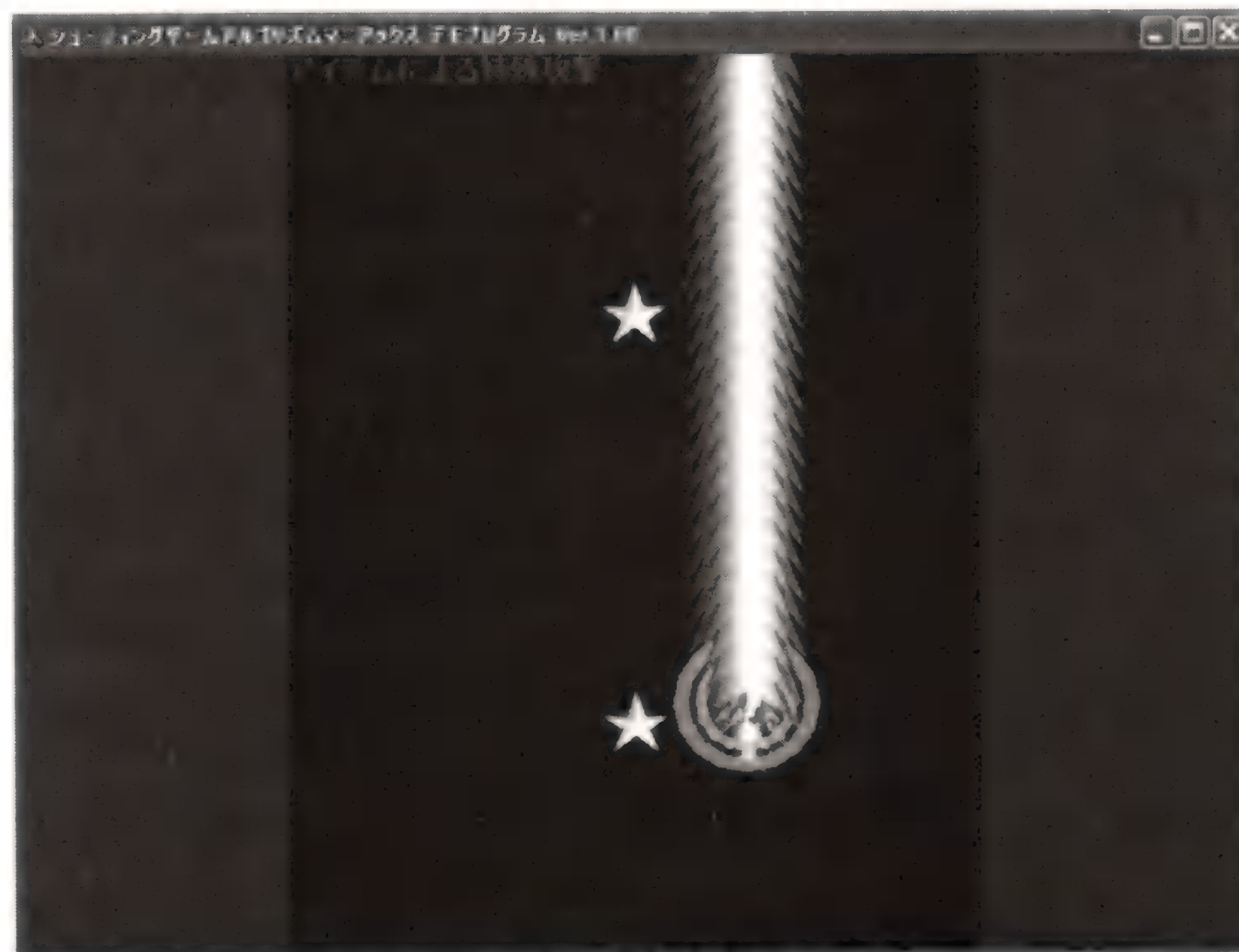
Fig. 5-8 特殊攻撃に関する状態遷移

特殊攻撃の流れをプログラムにまとめたのがList 5-4です。もしも特殊攻撃が複数種類あるときには、たとえば状態を「通常」「特殊1」「特殊2」「特殊3」のように分けて、それぞれ別々の攻撃処理を呼び出すようにします。

Fig. 5-9 特殊攻撃に関するフローチャート

サンプル

アイテムによる特殊攻撃 → P. 318

List 5-4 アイテムによる特殊攻撃

```
void ItemSpecialAttack(
    float x0, float y0,         // 自機の左上座標
    float x1, float y1,         // 自機の右下座標
    float ix0[], float iy0[],   // アイテムの左上座標
    float ix1[], float iy1[],   // アイテムの右下座標
    int num_item                // アイテムの数
) {
    static bool special=false;   // いまの状態が特殊ならばtrue
    static int time;             // 特殊攻撃の残り時間

    // 通常状態のとき
    if (!special) {

        // 通常攻撃を行う：
        // 具体的な処理はNormalAttack関数で行うとする。
        NormalAttack();
```

```
        // アイテムを拾ったかどうかの判定：
        // 拾ったら特殊状態に移り、残り時間を設定する。
        for (int i=0; i<num_item; i++) {
            if (ix0[i]<x1 && x0<ix1[i] &&
                iy0[i]<y1 && y0<iy1[i]) {
                special=true;
                time=300;
            }
        }
    }

    // 特殊状態のとき
    else {

        // 特殊攻撃を行う：
        // 具体的な処理はSpecialAttack関数で行うとする。
        SpecialAttack();

        // 一定時間が過ぎたかどうかの判定：
        // 残り時間がなくなったら通常状態に戻る。
        if (time==0) special=false; else time--;
    }
}
```

デモプログラム
Stage＃57　アイテムによる特殊攻撃

無敵状態

アイテムを拾ったりパワーが溜まったりすることによって、自機が敵や弾にやられなくなる状態です。無敵状態で敵にぶつかると敵にダメージを与えることができたり、弾にぶつかると弾を消すことができたりするゲームもあります (Fig. 5-10)。

通常状態 (無敵ではない状態) でも無敵状態でも、自機と敵や弾との間で当たり判定処理を行うことに変わりはありませんが、敵や弾にぶつかったあとの処理が異なります (Fig. 5-11)。通常状態では自機がやられてミスとなりますが、無敵状態では逆に敵にダメージを与えたり弾を消したりします。

List 5-5は無敵状態の処理を行うプログラムです。アイテムを拾うと無敵状態になるとか、一定時間が経つと通常状態に戻るといった処理は、「アイテムによる特殊攻撃」(→ P. 169) と同じ仕組みで実現することができます。

Fig. 5-10 無敵状態

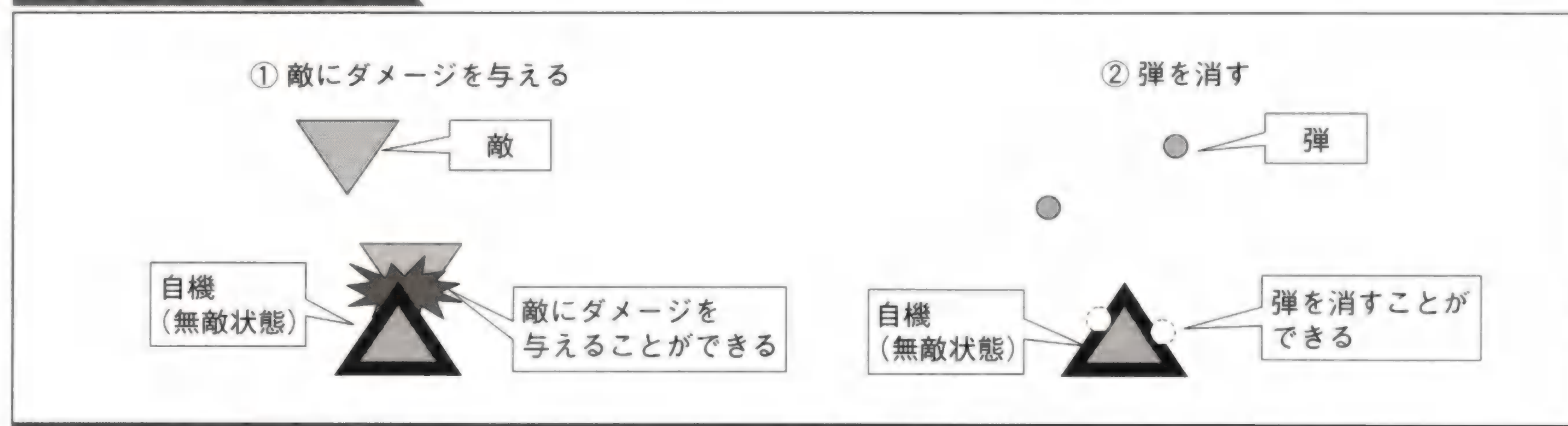



Fig. 5-11 敵や弾にぶつかったあとの処理

	通常状態	無敵状態
敵にぶつかる	ミス (または自機にダメージ)	敵にダメージ (または敵を破壊)
弾にぶつかる	ミス (または自機にダメージ)	弾を消す

サンプル

無敵状態 → P. 318

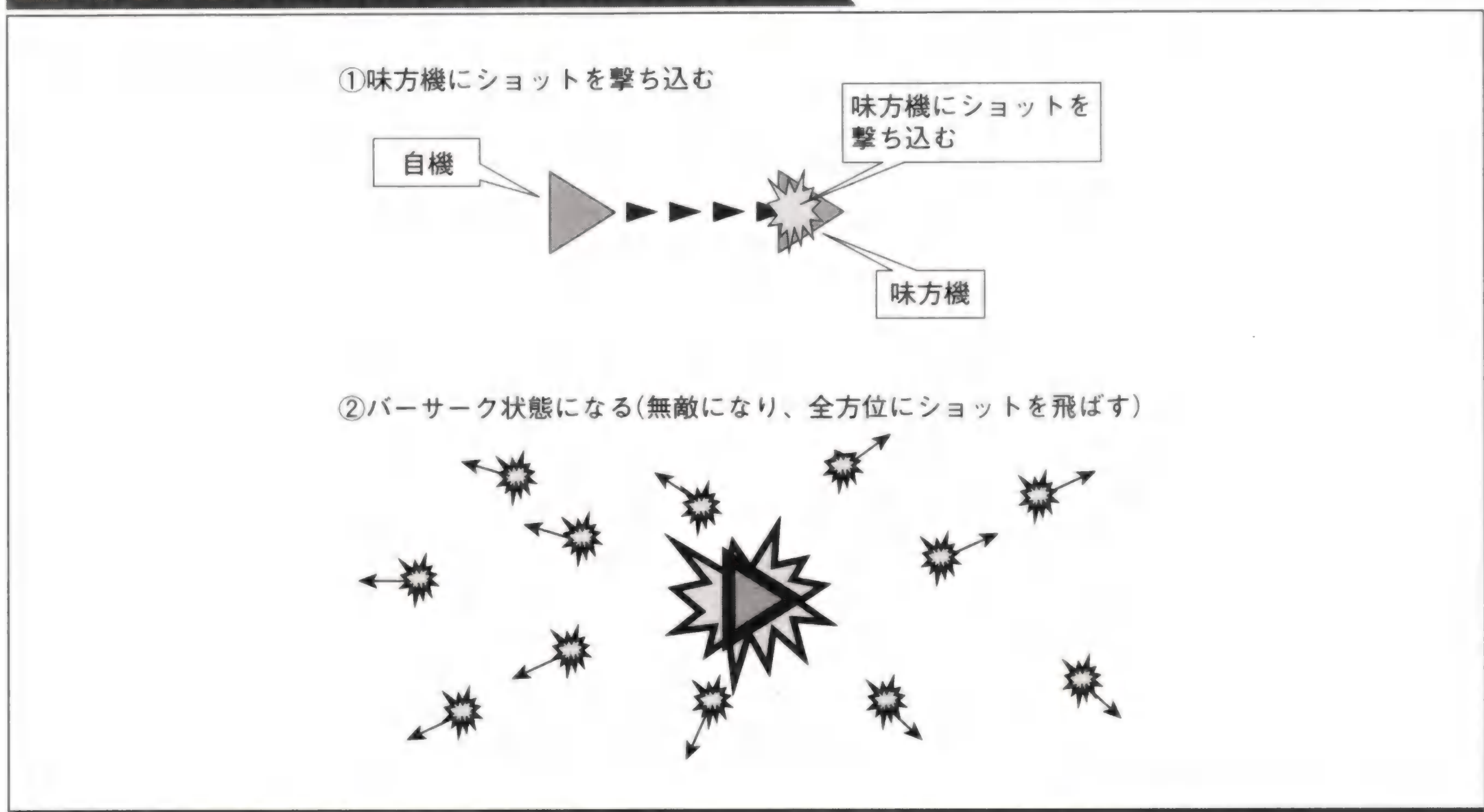
List 5-5 無敵状態

```
void Invincible(
    bool invincible,              // 無敵状態ならばtrue
    float x0, float y0,           // 自機の左上座標
    float x1, float y1,           // 自機の右下座標
    float ex0[], float ey0[],     // 敵の左上座標
    float ex1[], float ey1[],     // 敵の右下座標
    int num_enemy,                // 敵の数
    float bx0[], float by0[],     // 弾の左上座標
    float bx1[], float by1[],     // 弾の右下座標
    int num_bullet                // 弾の数
) {
    // 敵との当たり判定処理：
    // 敵とぶつかった場合、無敵状態ならば敵にダメージを与え、
    // 通常状態ならばミスとする。
    // ダメージ付与とミスの具体的な処理は、
    // DamageEnemy、Missの各関数で行うとする。
    for (int i=0; i<num_enemy; i++) {
        if (ex0[i]<x1 && x0<ex1[i] &&
            ey0[i]<y1 && y0<ey1[i]) {
            if (invincible) DamageEnemy(i); else Miss();
        }
    }

    // 弾との当たり判定処理：
    // 弾とぶつかった場合、無敵状態ならば弾を消し、
    // 通常状態ならばミスとする。
    // 消去とミスの具体的な処理は、
    // DeleteBullet、Missの各関数で行うとする。
    for (int i=0; i<num_bullet; i++) {
        if (bx0[i]<x1 && x0<bx1[i] &&
            by0[i]<y1 && y0<by1[i]) {
            if (invincible) DeleteBullet(i); else Miss();
        }
    }
}
```

バーサーク状態

「バーサーク（berserk）」という言葉はファンタジー小説やRPGなどでよく出てきますが、「狂暴な」というような意味です。特別な操作やアイテムなどによって自機が一定時間だけ大幅にパワーアップすることを、ここではバーサーク状態と呼ぶことにします。パワーアップの内容はさまざまですが、無敵になったり、強力なショットを全方位に発射したりといったことが考えられます。

バーサーク状態の面白い例は「極上パロディウス（→ P. 326）」の2人プレイです。「極上パロディウス」では、味方にショットをたくさん撃ち込むと、味方がバーサーク状態になります。この状態になった味方は一定時間無敵となるうえに、全方向に強力なショットを発射します（Fig. 5-12）。そのかわりにペナルティとして、バーサーク状態が終わったあとの一定時間は弾が撃てなくなります。

Fig. 5-12 「極上パロディウス」におけるバーサーク状態

バーサーク状態と通常状態との間の移行は、「アイテムによる特殊攻撃」（→ P. 169）における特殊状態と通常状態との間の移行に似ています（Fig. 5-13）。ここでは、バーサーク状態が終わったあとには弱い状態（弾が撃てないなどの状態）に移るものとしました。状態が3つになったぶん少し複雑ですが、「状態によって異なる処理に分岐する」という基本的な考え方は、状態が2つのときと同じです。ちなみに、Fig. 5-13の処理に相当するフローチャートはFig. 5-14のようになります。

List 5-6はバーサーク状態に関するプログラムです。シューティングゲームのプログラムでは、このように複数の状態の間を移行したり、状態に応じて処理を分岐したりといった処理をあちこちで使います。なお、List 5-6では敵や弾との当たり判定処理の詳細は省略しました。具体的な処理は「無敵状態」(→ P. 172) などのプログラムを参照してください。

Fig. 5-13 バーサーク状態に関する状態遷移

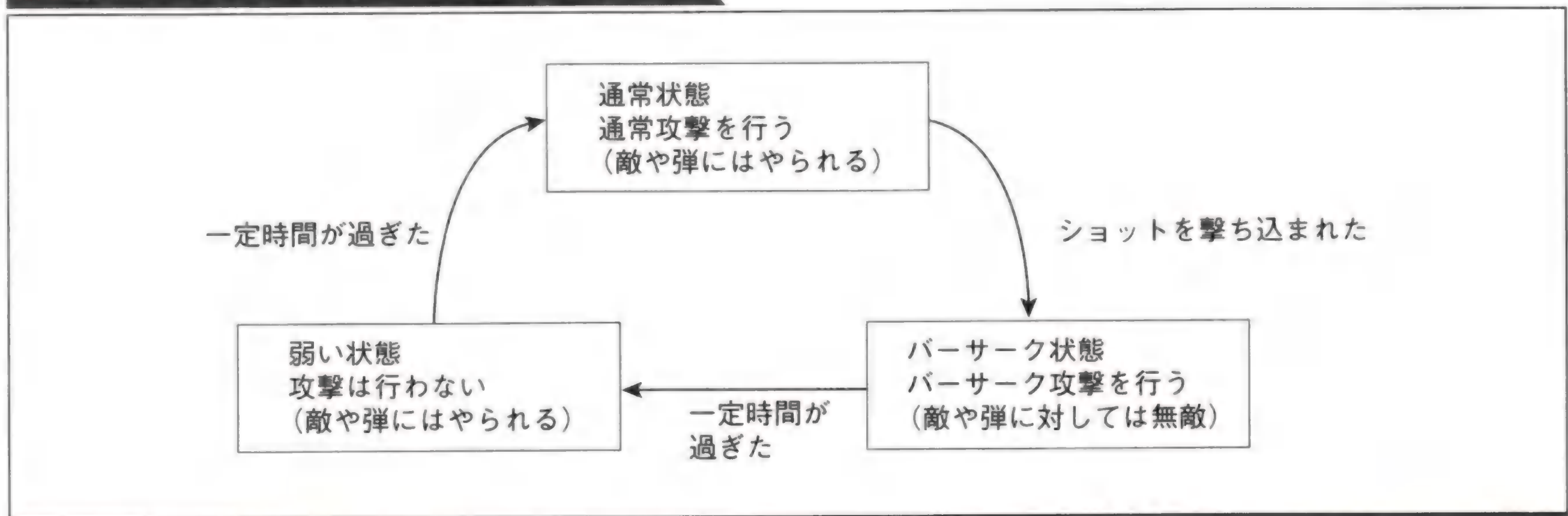
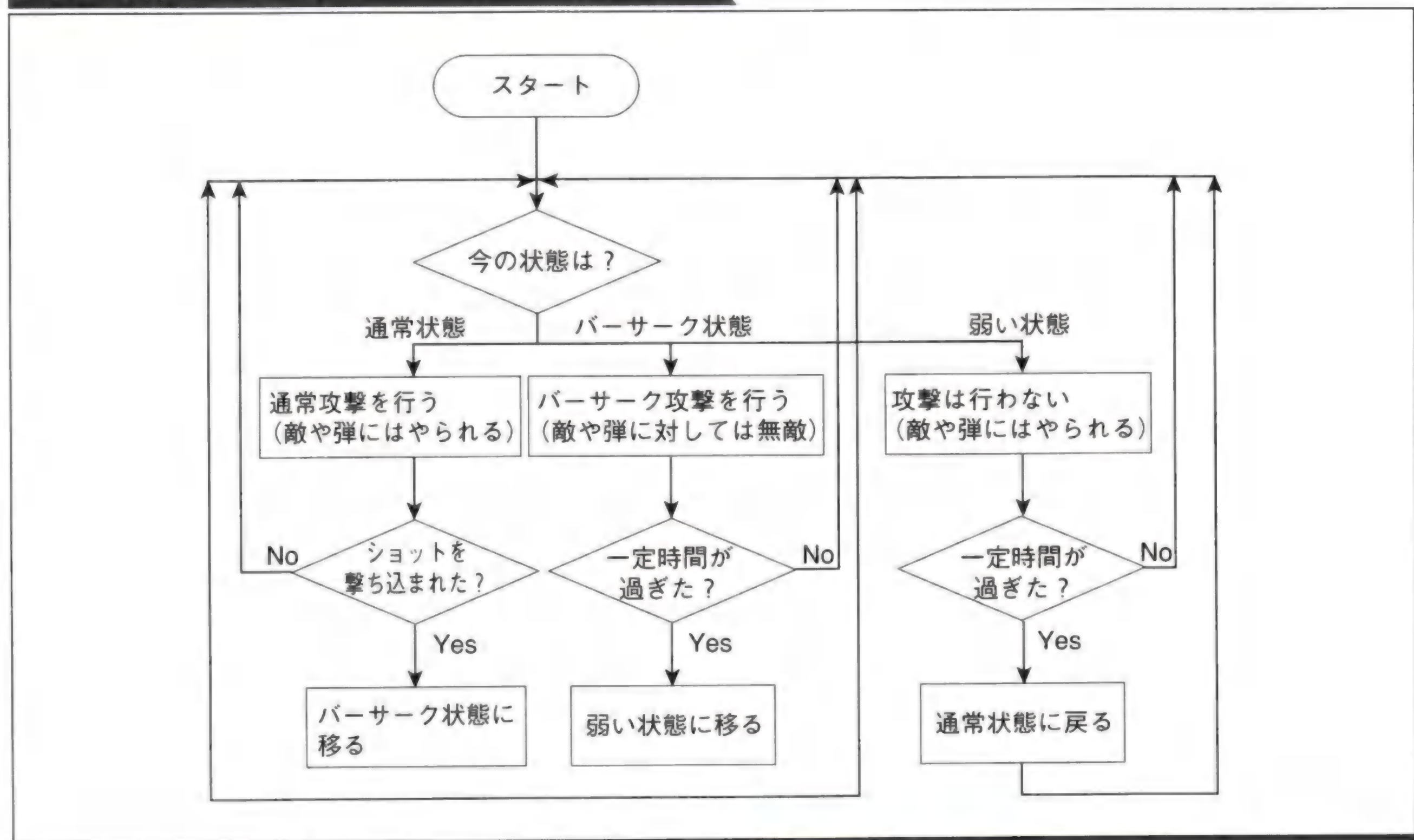



Fig. 5-14 バーサーク状態に関するフローチャート

サンプル

バーサーク →P. 318

List 5-6 バーサーク状態

```
// 自機の状態（通常、バーサーク、弱い）
typedef enum {
    NORMAL, BERSERK, WEAK
} STATE_TYPE;

// バーサーク状態の処理
void Berserk(
    int num_shot,   // 味方のショットの数
    int num_enemy,  // 敵の数
    int num_bullet  // 弾（敵弾）の数
) {
    static STATE_TYPE state=NORMAL;  // 自機の状態（最初は通常）
    static int energy=0;             // エネルギー
    static int berserk_energy=100;   // バーサーク状態の発動に
                                     // 必要なエネルギー
    static int time;                 // 効果の残り時間

    // 状態に応じて分岐する
    switch (state) {

        // 通常状態
        case NORMAL:

            // 通常攻撃：
            // 具体的な処理はNormalAttack関数で行うとする。
            NormalAttack();

            // 敵や弾との当たり判定処理：
            // 当たるとミスになる。
            // 判定や結果の具体的な処理はHitEnemy、
            // HitBullet、Missの各関数で行うとする。
            for (int i=0; i<num_enemy; i++)
                if (HitEnemy(i)) Miss();
            for (int i=0; i<num_bullet; i++)
                if (HitBullet(i)) Miss();

            // 味方のショットとの当たり判定処理：
            // エネルギーを溜める。
            // 判定や結果の具体的な処理はHitShot、
            // DeleteShotの各関数で行うとする。
            for (int i=0; i<num_shot; i++) {
                if (HitShot(i)) {
                    energy++;
                    DeleteShot(i);
                }
            }
```

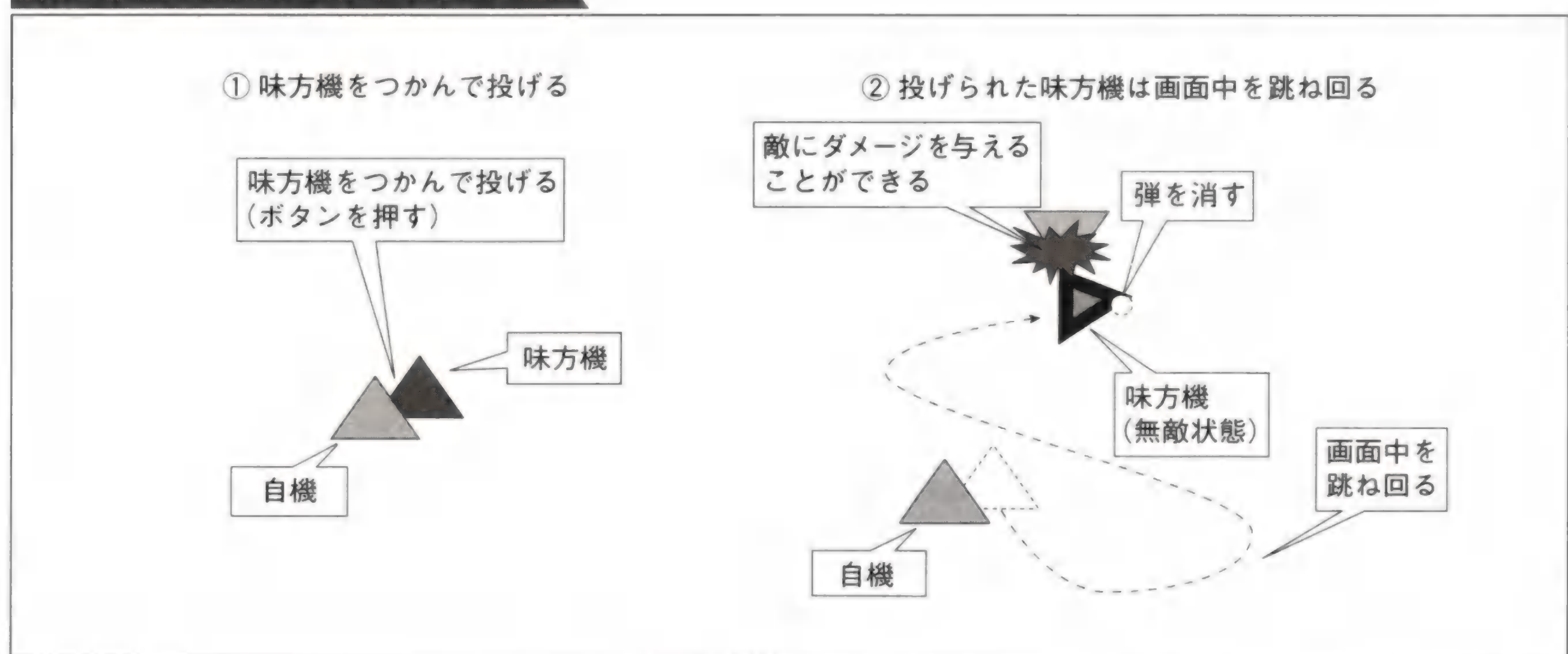
```
        // エネルギー量の判定：
        // エネルギーが十分に溜まったら
        // バーサーク状態に移行する。
        // エネルギーは自然に減っていく。
        if (energy>=berserk_energy) {
            state=BERSERK;
            time=300;
        } else {
            energy--;
        }

        break;

    // バーサーク状態
    case BERSERK:

        // 特殊攻撃：
        // 具体的な処理はSpecialAttack関数で行うとする。
        SpecialAttack();

        // 敵や弾との当たり判定処理：
        // 敵にダメージを与え、弾は消す。
        // 結果の具体的な処理はDamageEnemy、
        // DeleteBulletの各関数で行うとする。
        for (int i=0; i<num_enemy; i++)
            if (HitEnemy(i)) DamageEnemy(i);
        for (int i=0; i<num_bullet; i++)
            if (HitBullet(i)) DeleteBullet(i);

        // 残り時間が切れたら弱い状態に移行する
        time--;
        if (time<=0) {
            state=WEAK;
            time=200;
        }

        break;

    // 弱い状態
    case WEAK:

        // 敵や弾との当たり判定処理：
        // 当たるとミスになる。
        for (int i=0; i<num_enemy; i++)
            if (HitEnemy(i)) Miss();
        for (int i=0; i<num_bullet; i++)
```

```
            if (HitBullet(i)) Miss();

        // 残り時間が切れたら通常状態に移行する
        time--;
        if (time<=0) {
            state=NORMAL;
            energy=0;
        }

    break;
    }
}
```

味方をつかんで投げる

味方の近くでボタンを押すと、味方をつかんで投げることができるという攻撃です（Fig. 5-15）。味方は一定時間無敵になったまま画面内を跳ね回り、敵や弾を破壊します。この攻撃方法は「ツインビーヤッホー（→ P. 331）」などに見ることができます。

投げられた味方の状態は、「バーサーク状態」（→ P. 174）によく似ています。投げられたときの特徴は、味方は自分では機体のコントロールがいっさいできなくなることです。機体は画面枠のなかを跳ね回るように動きます。

Fig. 5-15 味方をつかんで投げる

画面枠で機体が跳ね返るようにするには、画面枠から機体がはみ出しそうになったときに、機体の移動速度を反転させます（Fig. 5-16）。移動速度を（vx, vy）とすると、左右で反転させるには次のようにします。

```
vx=-vx
```

上下で反転させる場合には次のようにします。

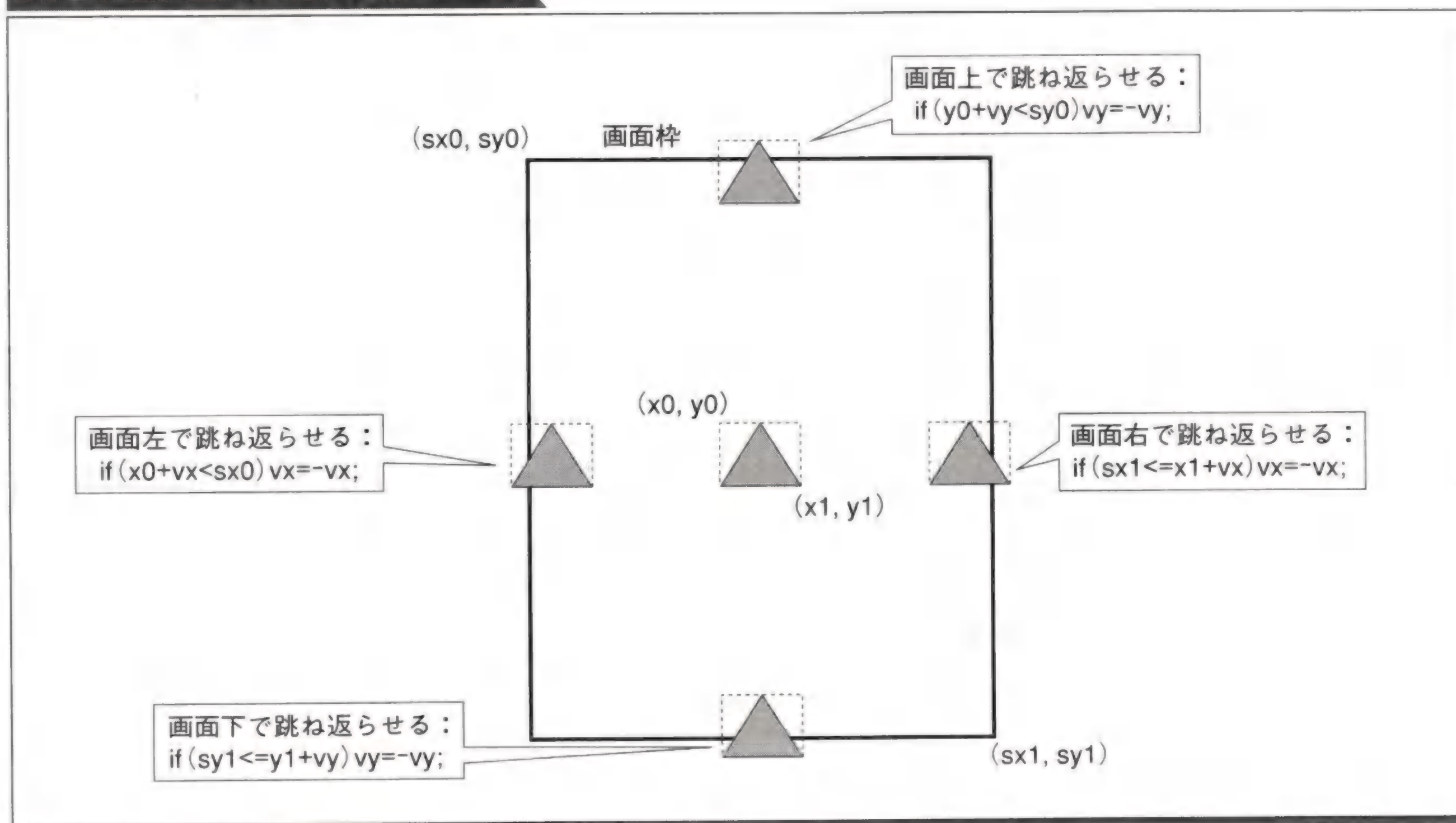
```
vy=-vy
```

List 5-7は投げられた機体の移動に関するプログラムです。機体が投げられたときには、普通にスティック入力で機体を移動するかわりに、List 5-7の処理を使って機体を移動します。

通常状態と投げられた状態との間の移行は、バーサーク状態と同じ要領で行えます。バーサーク状態は「味方にショットを当てること」で発動しますが、この発動条件を「味方の近くでボタンを押すこと」に変えればよいのです。

サンプル

味方をつかんで投げる　→ P. 318

Fig. 5-16　投げられた機体の移動

List 5-7 投げられた機体の移動

```
void MoveThrownShip(
    float& x0, float& y0,  // 機体の左上座標
    float& x1, float& y1,  // 機体の右下座標
    float& vx, float& vy,  // 機体の速度
    float sx0, float sy0,  // 移動可能範囲（画面枠）の左上座標
    float sx1, float sy1   // 移動可能範囲（画面枠）の右下座標
) {
    // 跳ね回る動きのための処理：
    // 画面枠の上下左右からはみ出しそうになったら、
    // 移動速度を逆にする。
    if (x0+vx<sx0 || sx1<=x1+vx) vx=-vx;
    if (y0+vy<sy0 || sy1<=y1+vy) vy=-vy;

    // 機体を移動させる
    x0+=vx; x1+=vx;
    y0+=vy; y1+=vy;
}
```

デモプログラム
Stage＃60　味方をつかんで投げる

敵をつかまえて投げる

敵をアームなどでとらえて、ボタン操作で投げつける攻撃です(Fig. 5-17〜5-19)。投げつけられた敵は爆発して、通常のショットよりも大きなダメージをほかの敵に与えます。「スペースボンバー(→ P. 328)」などは、この「敵をつかまえて投げる」という攻撃方法を中心としたゲームだといえます。「つかんで投げる」という動きは非常にコミカルなので、コミカルな雰囲気のゲームによく似合います。

「スペースボンバー」の場合、敵をとらえるにはアームを使います(Fig. 5-17)。アームの先端が敵に当たると、敵をとらえることができます。とらえた敵は自機の後ろについてきます(Fig. 5-18)。ここでボタンを押すと(「スペースボンバー」の場合は溜め撃ちの操作を行うと)敵を投げつけることができます(Fig. 5-19)。投げつけられた敵は爆発し、ほかの敵にダメージを与えます。

Fig. 5-17 敵をつかんでとらえる

敵
アーム
アームを伸ばして
敵をとらえる
自機

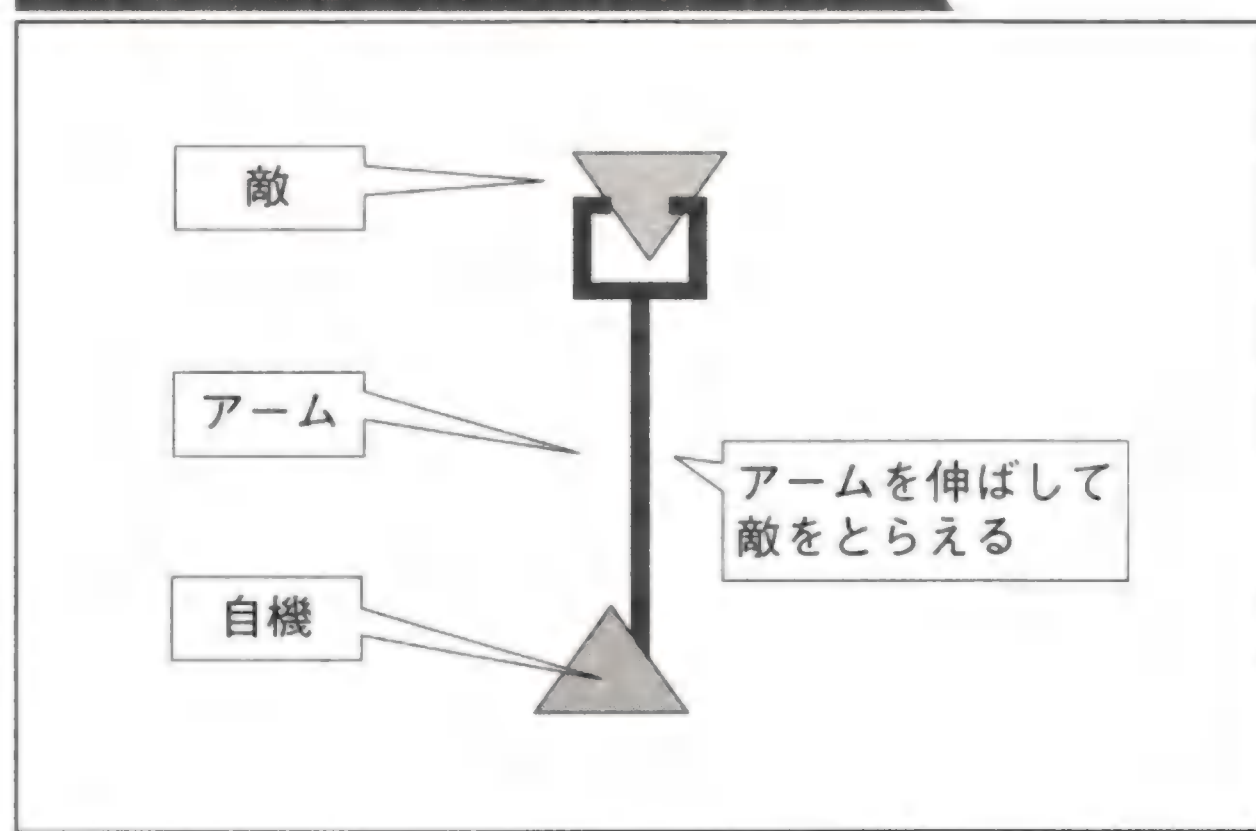
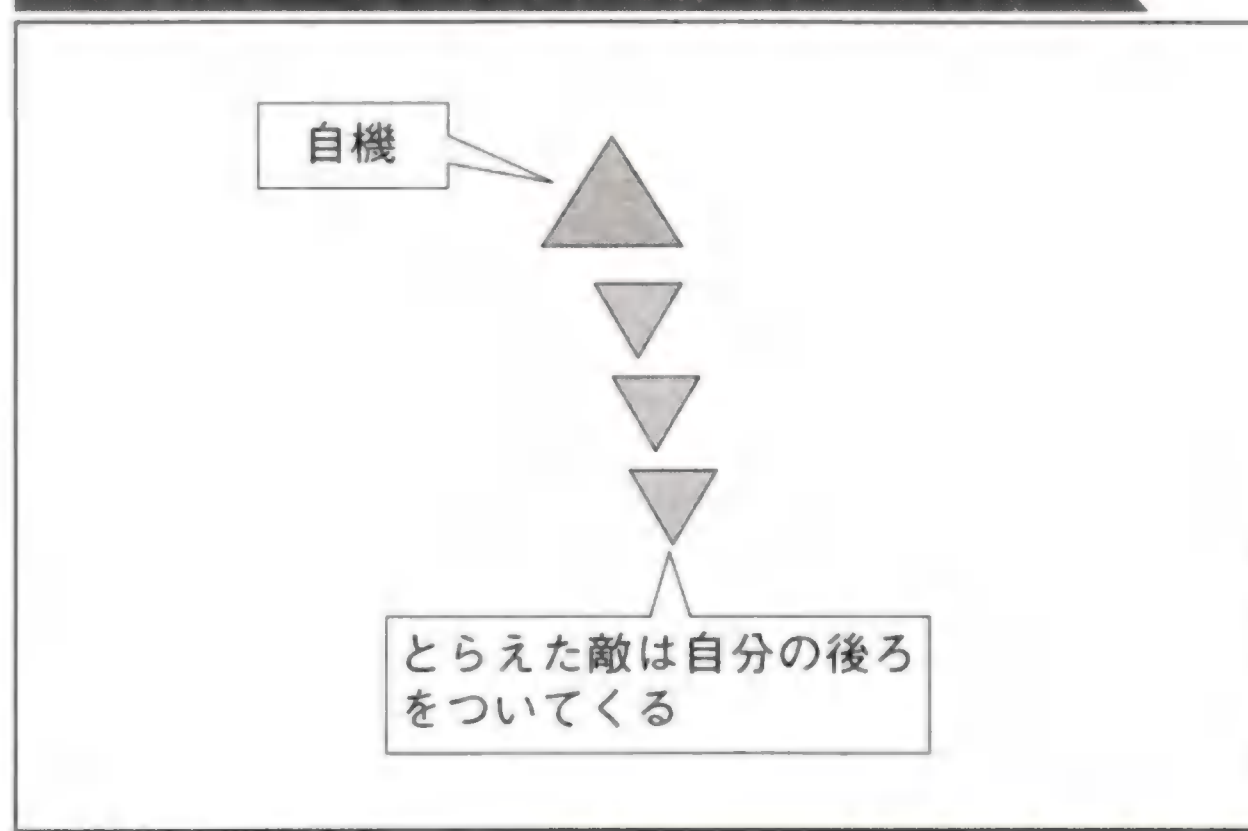
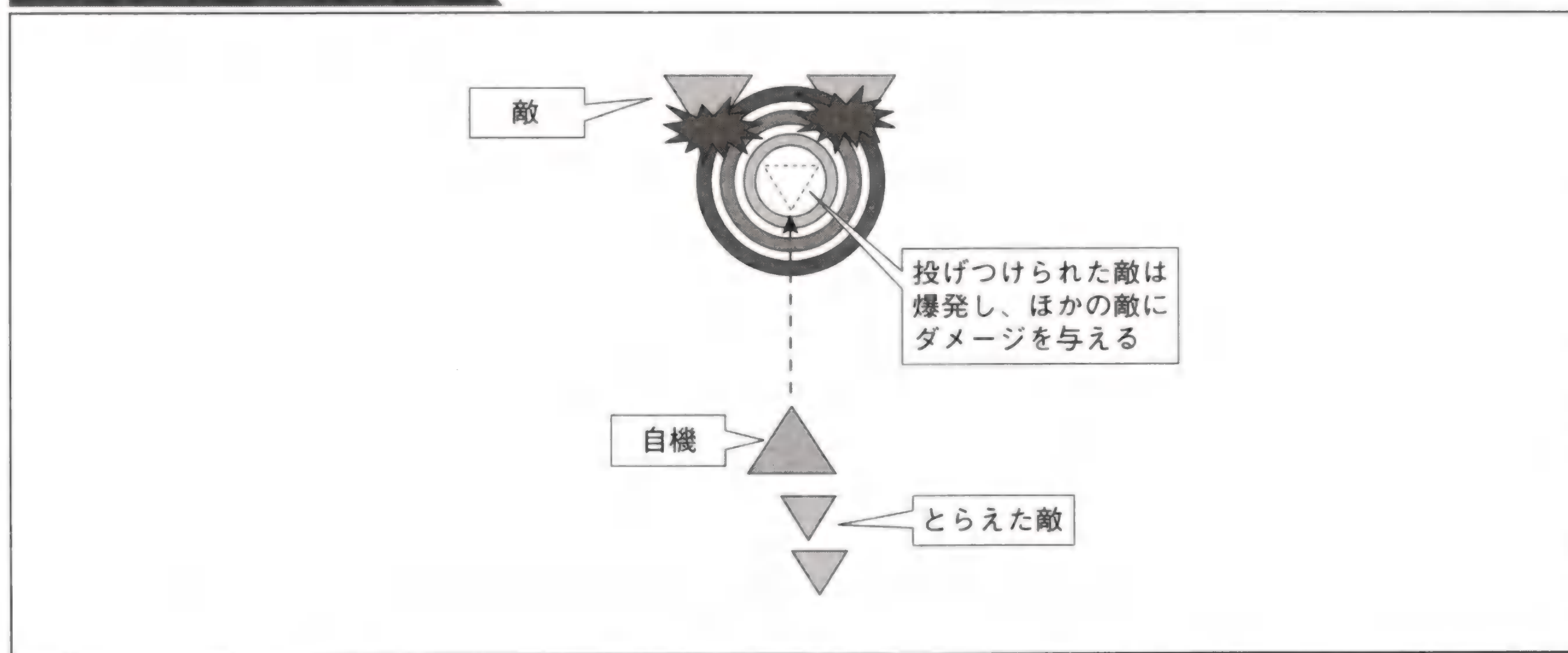



Fig. 5-18 とらえた敵は自機についてくる

自機
とらえた敵は自分の後ろ
をついてくる

Fig. 5-19 敵を投げつける

敵
投げつけられた敵は
爆発し、ほかの敵に
ダメージを与える
自機
とらえた敵

アームの動き

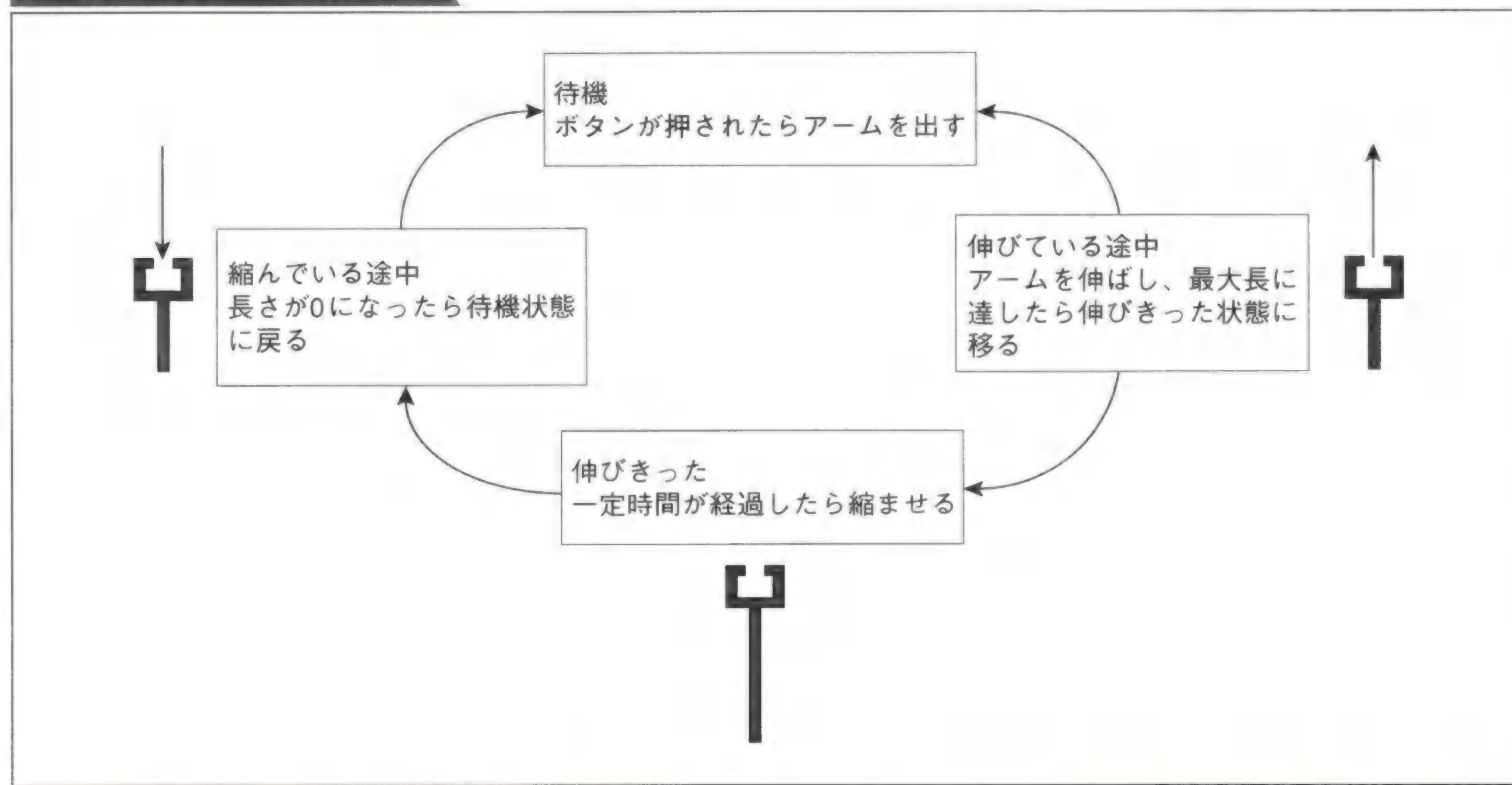
敵をつかんで投げる処理に関する第1のポイントは、アームの動きです（Fig. 5-20）。アームには「待機」「伸びている途中」「伸びきった」「縮んでいる途中」という4つの状態があるので、状態によって分岐して、それぞれの状態に応じた処理を行います。List 5-8はアームの動きに関する処理をまとめたプログラムです。

Fig. 5-20 アームの動き

待機
ボタンが押されたらアームを出す

伸びている途中
アームを伸ばし、最大長に達したら伸びきった状態に移る

伸びきった
一定時間が経過したら縮ませる

縮んでいる途中
長さが0になったら待機状態に戻る

List 5-8 アームの動き

```
// アームの状態：
// 待機、伸びている途中、伸びきった、縮んでいる途中。
typedef enum {
    READY, EXTENDING, EXTENDED, SHRINKING
} STATE_TYPE;

// アームの動き
void MoveArm(
    bool button   // アームを伸ばすボタンの状態
) {
    static STATE_TYPE state=READY;  // アームの状態
    static int length;              // アームの長さ
    static int max_length=20;       // アームの最大長
    static int time;                // 時間待ち用

    // アームの状態によって処理を変える
```

```
    switch (state) {

        // 待機:
        // ボタンが押されたらアームを伸ばす。
        case READY:
            if (button) {
                state=EXTENDING;
                length=0;
            }
            break;

        // 伸びている途中:
        // アームを伸ばし、
        // 最大長に達したら伸びきった状態に移る。
        case EXTENDING:
            if (length<max_length) {
                length++;
            } else {
                state=EXTENDED;
                time=10;
            }
            break;

        // 伸びきった:
        // 一定時間が経過したら縮ませる。
        case EXTENDED:
            if (time>0) {
                time--;
            } else {
                state=SHRINKING;
            }
            break;

        // 縮んでいる途中:
        // 長さが0になったら待機状態に戻る。
        case SHRINKING:
            if (length>0) {
                length--;
            } else {
                state=READY;
            }
            break;
    }
}
```

■ 投げつけられた敵の動き

第2のポイントは、投げつけられた敵の動きです。敵はショットのように飛んでいき、敵に当たると爆発して、ダメージを与えます。投げつけられた敵とすべての敵との間で当たり判定処理を行って、もしも衝突したら、投げつけられた敵を消し、かわりに爆風を出現させます。爆風を出したあとの処理は「誘爆」(→ P. 167) と同じ要領です。投げつけられた敵の動きに関する処理はList 5-9にまとめました。

サンプル

敵をつかんで投げる → P. 318

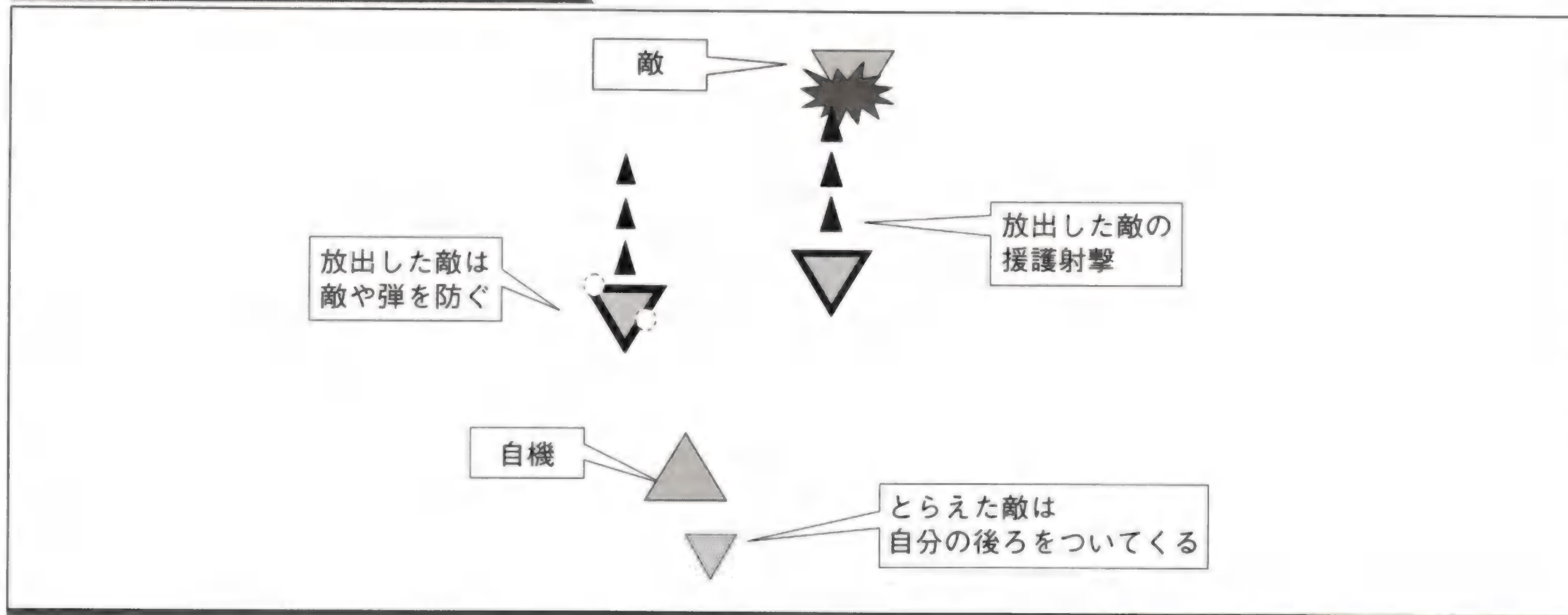
List 5-9 投げつけられた敵の動き

```
void MoveThrownEnemy(
    float& x0, float& y0,      // 投げつけられた敵の左上座標
    float& x1, float& y1,      // 投げつけられた敵の右下座標
    float vx, float vy,        // 投げつけられた敵の速度
    float ex0[], float ey0[],  // ほかの敵の左上座標
    float ex1[], float ey1[],  // ほかの敵の右下座標
    int num_enemy              // ほかの敵の数
) {
    // 敵との当たり判定処理：
    // 投げつけられた敵と、ほかのすべての敵との間で
    // 当たり判定処理を行う。
    // 衝突したら、投げつけられた敵を消し、
    // かわりに爆風を出す。
    // 消滅と出現の具体的な処理は、
    // DeleteEnemy、CreateExplosionの各関数で行うとする。
    int i;
    for (i=0; i<num_enemy; i++) {
        if (ex0[i]<x1 && x0<ex1[i] &&
            ey0[i]<y1 && y0<ey1[i]) {
            DeleteEnemy(i);
            CreateExplosion(x0, y0);
        }
    }

    // どの敵にもぶつからなかった場合：
    // 投げつけられた敵を動かす。
    if (i==num_enemy) {
        x0+=vx; y0+=vy;
        x1+=vx; y1+=vy;
    }
}
```

敵をつかまえて味方にする

敵をアームやビームなどでとらえて味方にして、自機を援護させる攻撃です。味方になった敵は一定のダメージが蓄積して消えるまで、援護射撃をしたり自機の盾となって敵や弾から守ってくれたりします。この攻撃方法は「スペースボンバー（→ P. 328）」「ダライアス外伝」（→ P. 330）「Gダライアス（→ P. 328）」などに見られます。

「スペースボンバー」の場合、敵をとらえるところまでは「敵をつかまえて投げる」（→ P. 181）と同じ操作です。とらえた敵を特定の操作で放出すると味方になります（Fig. 5-21）。味方になった敵は援護射撃をするほか、敵や弾を防いでくれます。ただし一定値以上のダメージが蓄積すると消えてしまいます。

Fig. 5-21　味方になった敵の動き

味方になった敵の性質は、敵や自機の性質と比べてみるとよくわかります（Fig. 5-22）。味方になった敵は自機と同じように、敵に対してダメージを与えるショットを撃ちます。敵や弾に当たったときにはダメージを受けますが、自機が当たったときのようにミスになることはありません。

味方になった敵の動きをプログラムにしたものがList 5-10です。ここでは敵も弾もダメージを一律に1としましたが、敵や弾の強さに応じてダメージを変えても面白いでしょう。

サンプル

敵をつかまえて味方にする　→ P. 319

Fig. 5-22 味方になった敵の性質

	通常の敵	味方になった敵	自機
敵にぶつかる	影響なし	ダメージを受けながら 敵にもダメージを与える （ミスにはならない）	ミスになる
弾にぶつかる	影響なし	ダメージを受けながら 弾を消す （ミスにはならない）	ミスになる
撃つショット（弾）の性質	自機を破壊する	敵にダメージを与える	敵にダメージを与える

List 5-10 味方になった敵の動き

```
void CapturedEnemy(
    float& x0, float& y0,       // 味方になった敵の左上座標
    float& x1, float& y1,       // 味方になった敵の右下座標
    float ex0[], float ey0[],   // ほかの敵の左上座標
    float ex1[], float ey1[],   // ほかの敵の右下座標
    int num_enemy,              // ほかの敵の数
    float bx0[], float by0[],   // 弾の左上座標
    float bx1[], float by1[],   // 弾の右下座標
    int num_bullet,             // 弾の数
    bool button                 // ショットボタンの状態
) {
    static int damage;          // 現在のダメージ
    static int max_damage=40;   // ダメージの限界値

    // 敵との当たり判定処理：
    // 敵と衝突したら、敵にダメージを与え、
    // 自分もダメージを受ける。
    // ダメージ付与の具体的な処理は、
    // DamageEnemy関数で行うとする。
    for (int i=0; i<num_enemy; i++) {
        if (ex0[i]<x1 && x0<ex1[i] &&
            ey0[i]<y1 && y0<ey1[i]) {
            DamageEnemy(i);
            damage++;
        }
    }

    // 弾との当たり判定処理：
```

```
    // 弾と衝突したら、弾を消し、
    // 自分もダメージを受ける。
    // 消去の具体的な処理は、
    // DeleteBullet関数で行うとする。
    for (int i=0; i<num_bullet; i++) {
        if (bx0[i]<x1 && x0<bx1[i] &&
            by0[i]<y1 && y0<by1[i]) {
            DeleteBullet(i);
            damage++;
        }
    }

    // 援護射撃：
    // 自機のショット操作に合わせてショットを撃つ。
    // 発射の具体的な処理はShot関数で行うとする。
    // なお、ボタン操作をしなくても
    // 自動的に援護射撃を行うようにする方法もある。
    if (button) Shot();

    // ダメージ判定：
    // ダメージが限界値まで蓄積したら消滅する。
    // 消滅の具体的な処理は、
    // DeleteCapturedEnemy関数で行うとする。
    if (damage>=max_damage) DeleteCapturedEnemy();
}
```

デモプログラム
Stage＃62　敵をつかまえて味方にする

◉敵につかまった自機を取り返してパワーアップする

敵に一度つかまった自機を、あとから取り返すという攻撃方法です。取り返した自機は味方となり、自機の攻撃がパワーアップします。これは「ギャラガ（→ P. 325）」における非常にユニークな要素です。

「ギャラガ」の場合、まず自機が敵につかまる（自機を敵につかまえさせる）ところから始まります。敵が放つビームに接触すると自機がとらえられ、捕虜になってしまいます（Fig. 5-23）。この自機をとらえた敵をうまく狙って破壊すると、自機を取り返すことができます（Fig. 5-24）。取り返した自機は自機の横に並び、ショットが2連装になることによって攻撃力が倍増します（Fig. 5-25）。

この攻撃手段で重要な役割を果たすのは敵です。自機をつかまえる敵には2つの状態があります（Fig. 5-26）。最初は通常状態で、このとき敵は通常攻撃か捕捉攻撃（自機をとらえる攻撃）をします。自機をつかまえると捕捉状態に移り、このときは通常攻撃のみになります。もしも捕捉状態で破壊されたら、とらえた自機を返してから消滅します。

List 5-11は自機をつかまえる敵の動きをまとめたプログラムです。

Fig. 5-23 自機が敵につかまる

敵
ビームに自機が接触すると敵にとらえられてしまう
ビーム
自機

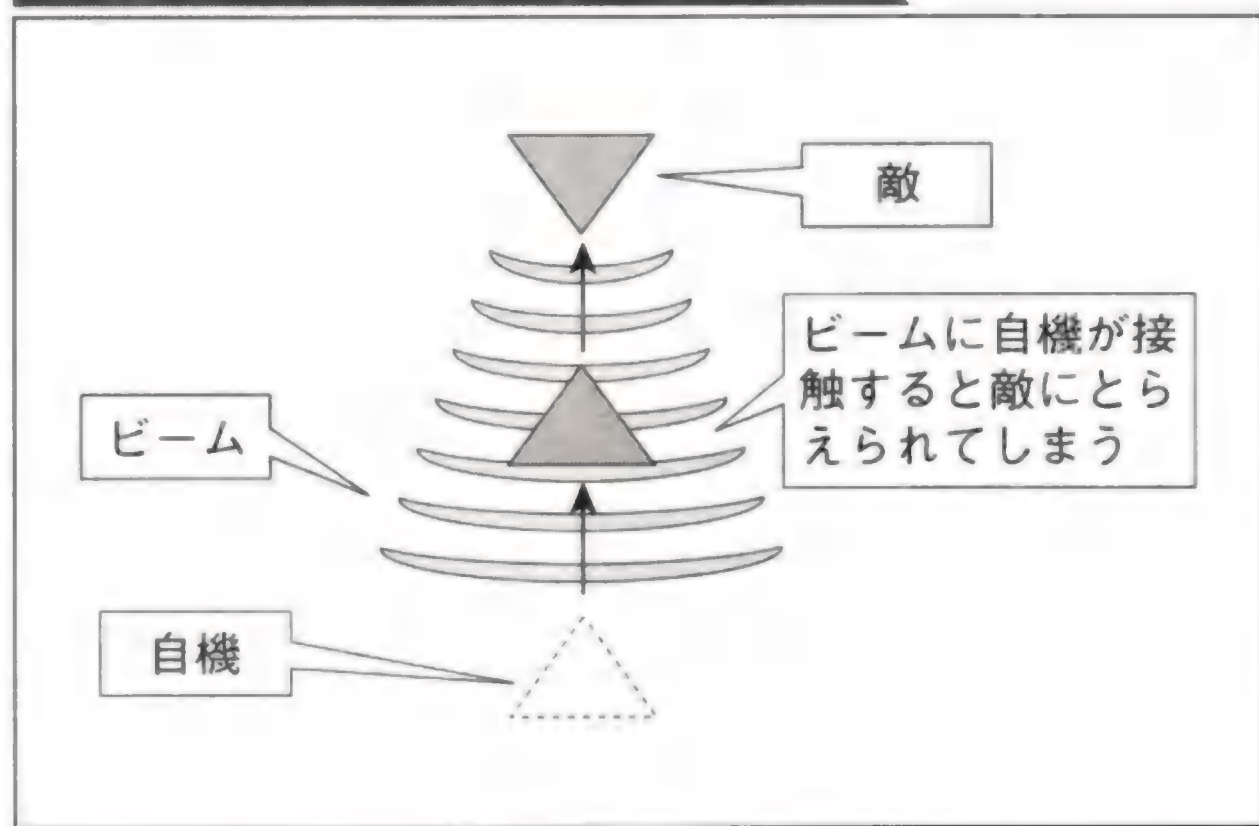
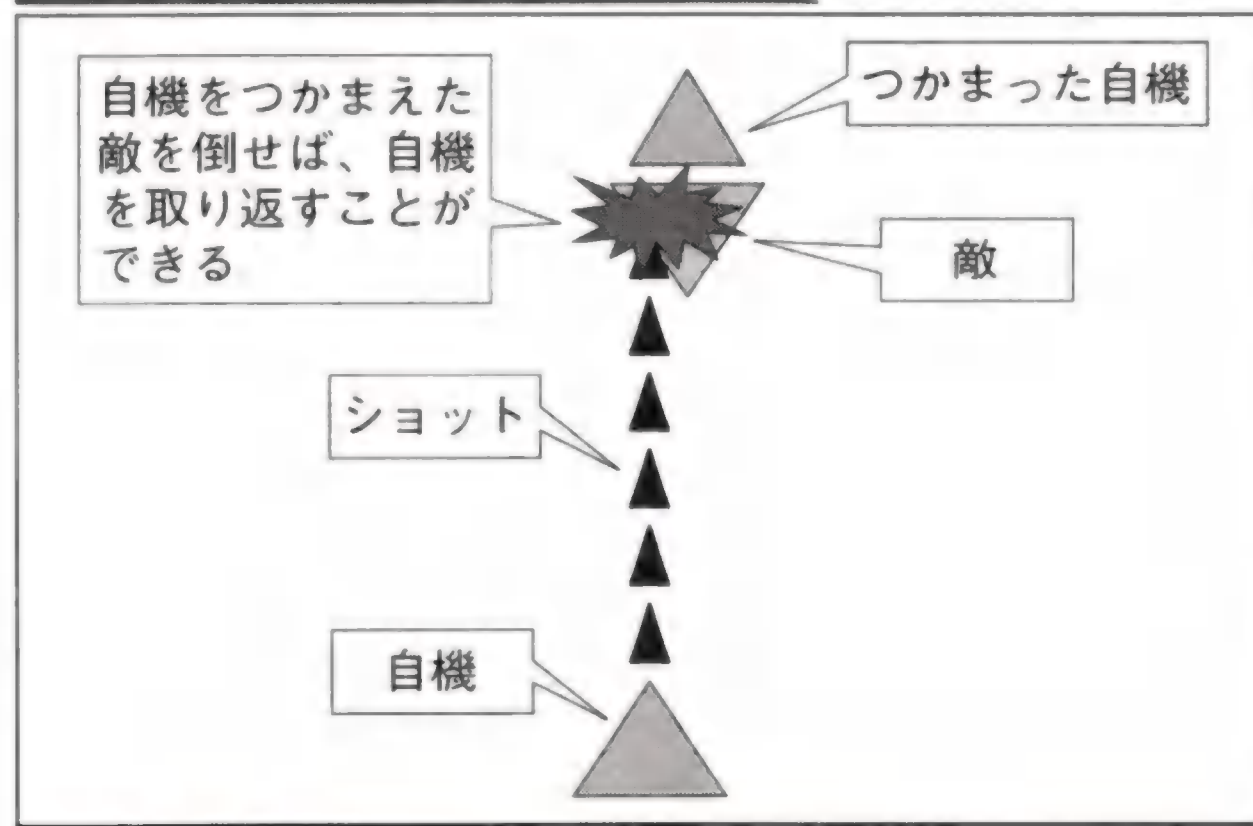
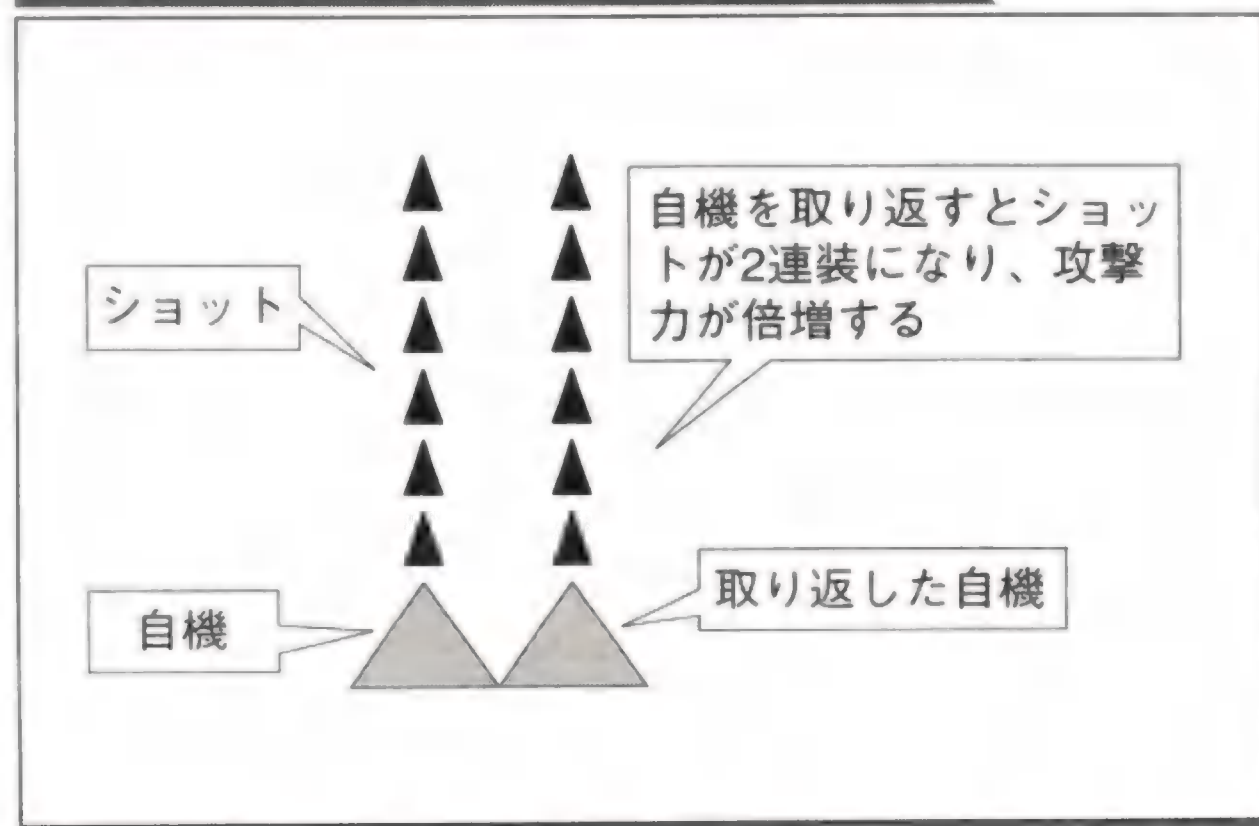
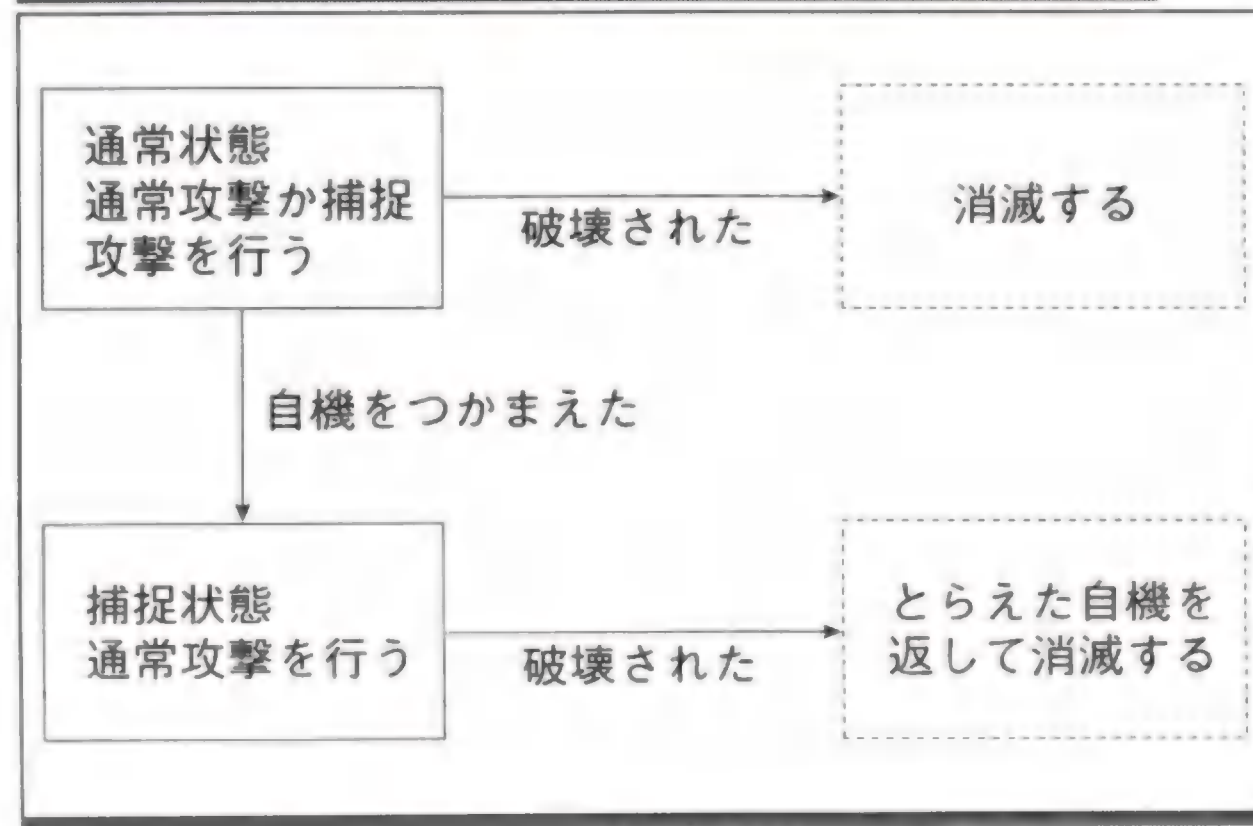
Fig. 5-24 自機を取り返す

自機をつかまえた敵を倒せば、自機を取り返すことができる
つかまった自機
敵
ショット
自機

Fig. 5-25 ショットが2連装になる

自機を取り返すとショットが2連装になり、攻撃力が倍増する
ショット
自機
取り返した自機

Fig. 5-26 自機をつかまえる敵の状態遷移

通常状態
通常攻撃か捕捉攻撃を行う
破壊された
消滅する
自機をつかまえた
捕捉状態
通常攻撃を行う
破壊された
とらえた自機を返して消滅する

サンプル

敵につかまった自機を取り返してパワーアップする → P. 319

List 5-11 自機をつかまえる敵の動き

```
// 敵の状態：
// 通常、捕捉
typedef enum {
    NORMAL, CAPTURED
} CAPTOR_STATE;

// 自機をとらえる敵の動き
void Captor(
    bool capture  // 今回、捕捉攻撃をするかどうか
) {
    static int state=NORMAL;  // 敵の状態

    // 状態によって分岐する
    switch (state) {

        // 通常状態：
        // 通常攻撃または捕捉攻撃を行う。
        // 捕捉攻撃が成功したら捕捉状態に移る。
        // 攻撃の具体的な処理は、CaptureAttack、NormalAttackの各関数で行うとする。
        case NORMAL:
            if (capture) {
                if (CaptureAttack()) state=CAPTURED;
            } else {
                NormalAttack();
            }
            break;

        // 捕捉状態：
        // 通常攻撃を行う。
        // 破壊されたら自機を返したあとに消滅する。
        // 判定などの具体的な処理は、Destroyed、ReturnMyShip、Deleteの各関数で行うとする。
        case CAPTURED:
            NormalAttack();
            if (Destroyed()) {
                ReturnMyShip();
                Delete();
            }
            break;
    }
}
```

味方に接近してショットを強化する

味方に近づいてショットを撃つと、ショットが通常よりも強力になるというものです（Fig. 5-27）。この攻撃方法は「出たな!!ツインビー（→ P. 331）」「ツインビーヤッホー（→ P. 331）」などで見ることができます。

味方に接近したかどうかを判断するには、自機と味方との当たり判定処理を行います。このときの当たり判定は少し大きめにしておくのがよいでしょう（Fig. 5-28）。自機も味方も絶えず動いているので、ある程度離れていても「接近した」と判断されるほうが、プレイヤーにとって遊びやすくなるはずです。

自機の左上座標を（x0, y0）、右下座標を（x1, y1）、味方の左上座標を（cx0, cy0）、右下座標を（cx1, cy1）とすると、当たり判定が重なる条件は次のようになります。

(cx0<x1 && x0<cx1 && cy0<y1 && y0<cy1)

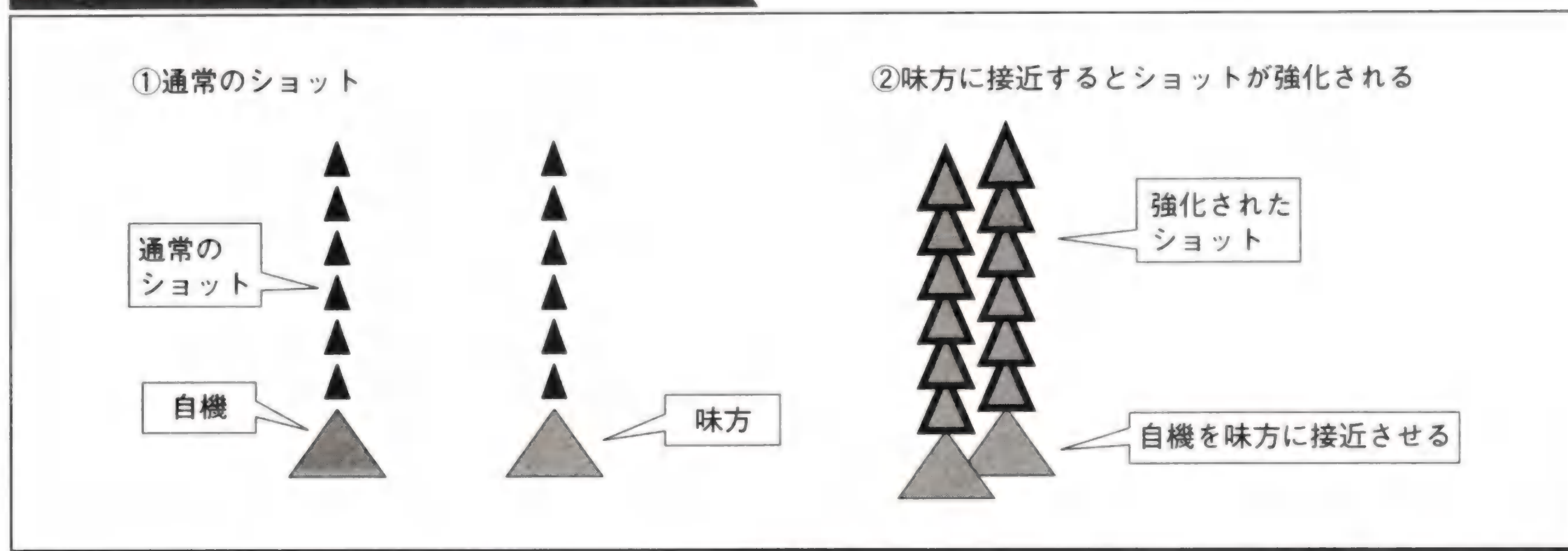
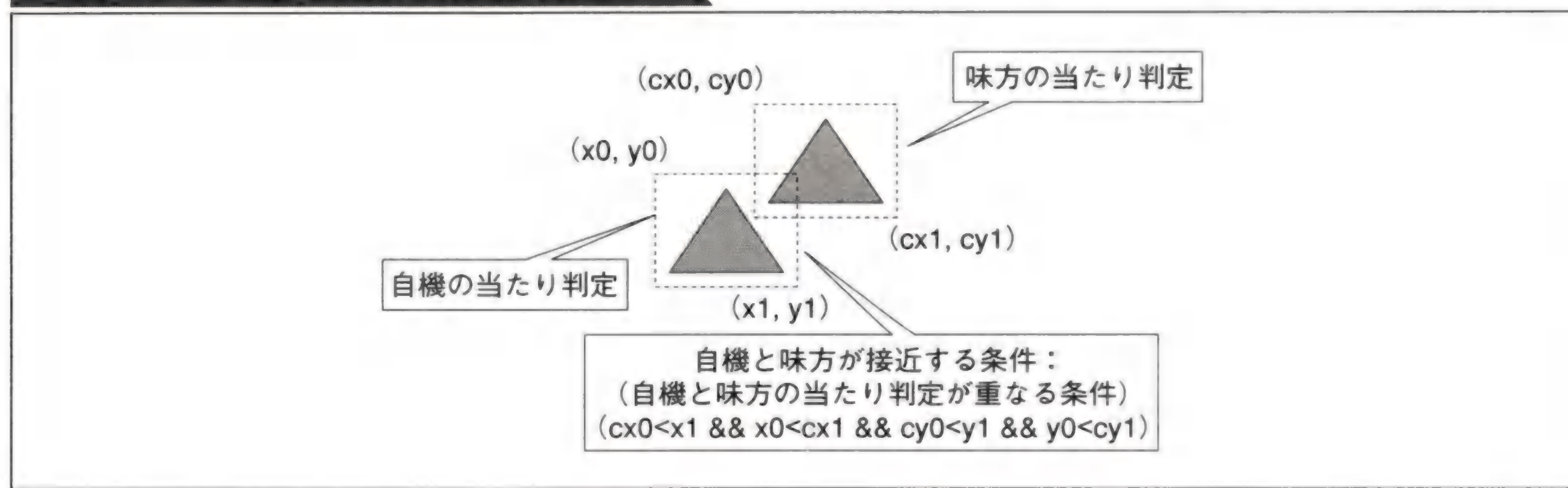
Fig. 5-27 味方に接近してショットを強化する

①通常のショット　②味方に接近するとショットが強化される

通常のショット　自機　味方　強化されたショット　自機を味方に接近させる

Fig. 5-28 味方に接近したかどうかの判定

(cx0, cy0)　味方の当たり判定　(x0, y0)　(cx1, cy1)　自機の当たり判定　(x1, y1)

自機と味方が接近する条件：
（自機と味方の当たり判定が重なる条件）
(cx0<x1 && x0<cx1 && cy0<y1 && y0<cy1)

味方と接近したときにショットを強化する処理はList 5-12のようになります。もしも味方が2機以上いる場合には、すべての味方との当たり判定処理を行い、その結果に応じてショットを強化します。

ちなみに「味方と合体する」という攻撃方法も同じ要領で実現できます。たとえば自機と味方が非常に接近したときには合体扱いにして、特別なグラフィックやエフェクトを出せばよいのです。

サンプル

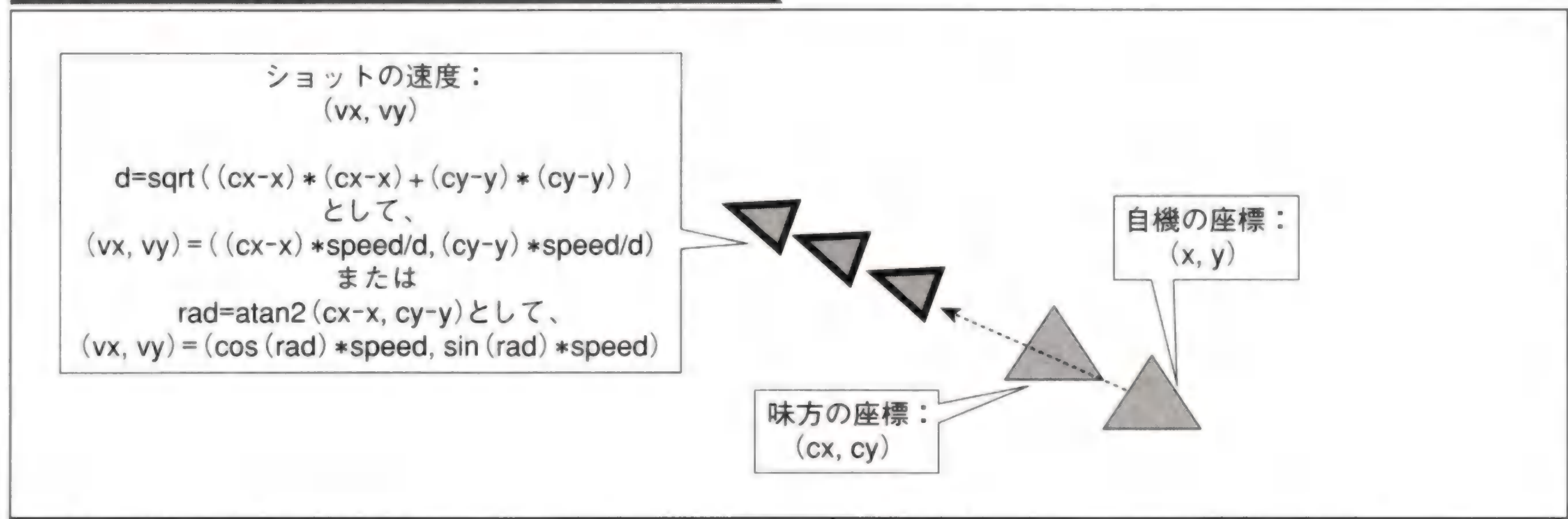
味方に接近してショットを強化する →P. 319

List 5-12 味方と接近してショットを強化する

```
void ReinforcedShot(
    float x0, float y0,      // 自機の左上座標
    float x1, float y1,      // 自機の右下座標
    float cx0, float cy0,    // 味方機の左上座標
    float cx1, float cy1,    // 味方機の右下座標
    bool button              // ショットボタンの状態
) {
    // 味方と接近しているかどうかの判定：
    // 接近している場合には強いショット、
    // していない場合には通常のショットを撃つ。
    // ショットを撃つ具体的な処理は、
    // SpecialShot、NormalShotの各関数で行うとする。
    if (button) {
        if (cx0<x1 && x0<cx1 && cy0<y1 && y0<cy1) {
            SpecialShot();
        } else {
            NormalShot();
        }
    }
}
```

味方がいる方向に強いショットを撃つ

味方と自機との位置関係によってショットが出る方向を変えるというのも面白い攻撃方法です。たとえば「ツインビーヤッホー（→ P. 331）」では、味方と自機が接近したとき、味方がいる方向に強いショットが発射されます（Fig. 5-29）。

Fig. 5-29 味方がいる方向に強いショットを撃つ

自機の座標を（x, y）、味方の座標を（cx, cy）とすると、ショットの速度（vx, vy）は次のように求められます。

```
d=sqrt((cx-x)*(cx-x)+(cy-y)*(cy-y))
vx=(cx-x)*speed/d
vy=(cy-y)*speed/d
```

あるいは、次のようにしても結果は同じです。

```
rad=atan2(cy-y, cx-x)
vx=cos(rad)*speed
vy=sin(rad)*speed
```

プログラムにまとめるとList 5-13のようになります。

サンプル

味方がいる方向に強いショットを打つ　→ P. 319

List 5-13 味方がいる方向に強いショットを撃つ

```
#include <math.h>

void ReinforcedShot2(
    float x, float y,     // 自機の座標
    float cx, float cy,   // 味方機の座標
    float speed           // ショットの速さ
) {
    // ショットの速度を求める
    float vx=cx-x, vy=cy-y;
    float d=sqrt(vx*vx+vy*vy);
    vx*=speed/d;
    vy*=speed/d;

    // ショットを発射する：
    // 発射の具体的な処理はSpecialShot関数で行うとする。
    SpecialShot(x, y, vx, vy);
}
```

味方に当てたショットを強化する

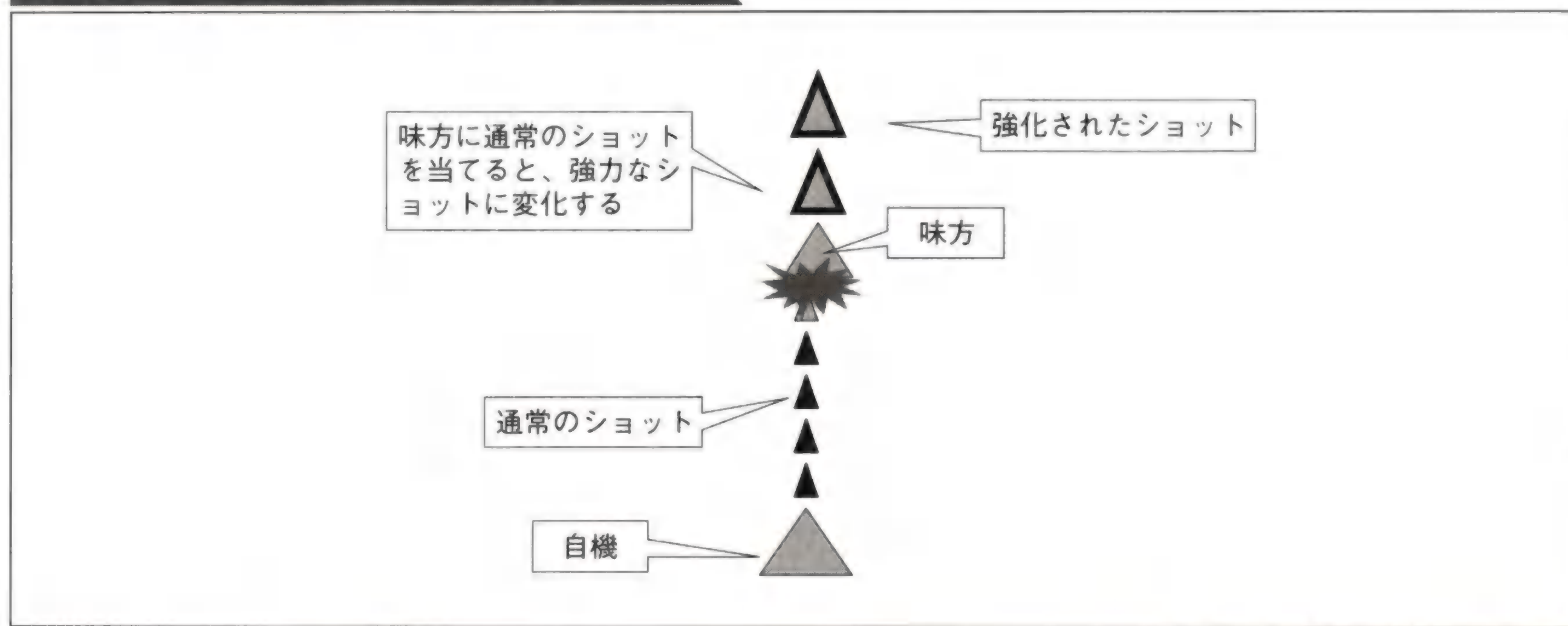
味方にショットを当てると、もっと強力なショットに変わるという攻撃方法です(Fig. 5-30)。「味方に接近してショットを強化する」(→ P. 190) に似ていますが、こちらは味方と自機が少し離れているときにもショットが強化されます。この攻撃方法は「ツインビーヤッホー (→ P. 331)」などに見られます。

基本的には、ショットが味方に当たったときに強力なショットに変化させればよいのです。ショットと味方との当たり判定処理を行い、当たったときにはそのショットを消して、かわりに強力なショットを出します。プログラムはList 5-14のようになります。

「ショットを消して強力なショットを出す」というかわりに、「ショットの属性を通常から強力に変える」という処理方法もあります。結果は同じなので、プログラムが書きやすい方法を選ぶとよいでしょう。

サンプル

● 味方に当てたショットを強化する　→ P. 319

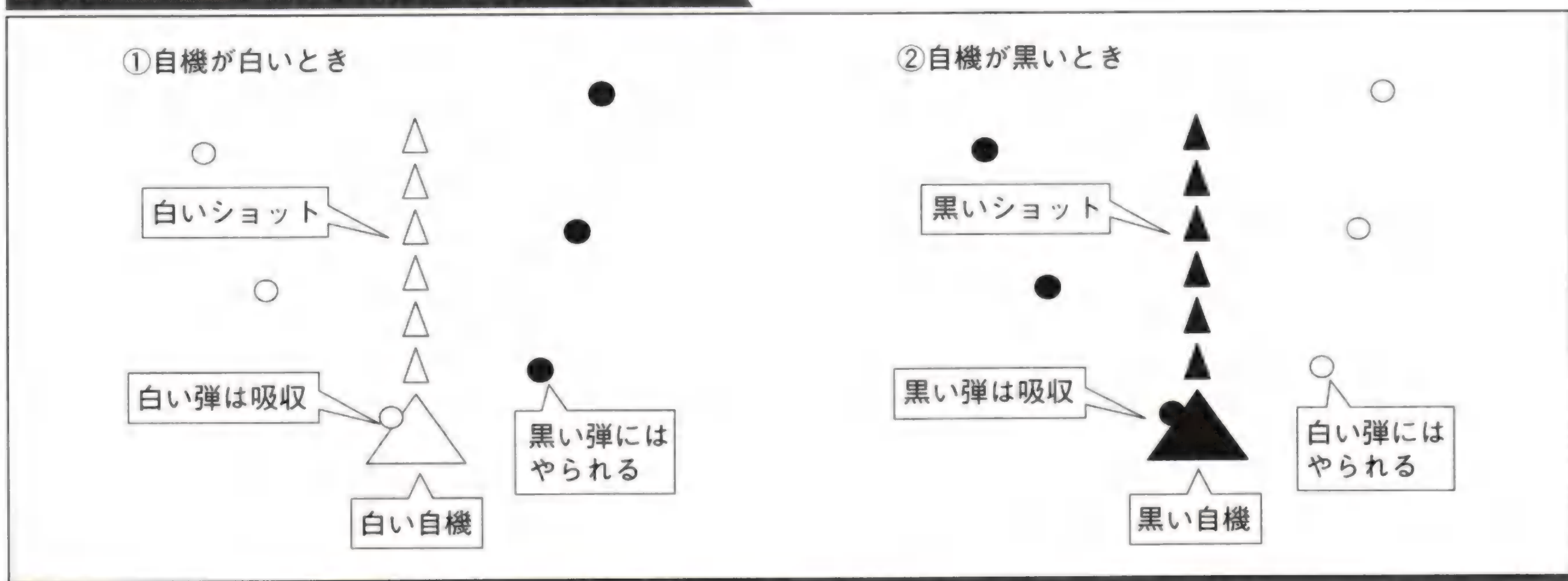
Fig. 5-30 味方に当てたショットを強化する

List 5-14 味方に当てたショットを強化する

```
void ReinforcedShot3(
    float sx0, float sy0,  // ショットの左上座標
    float sx1, float sy1,  // ショットの右下座標
    float cx0, float cy0,  // 味方の左上座標
    float cx1, float cy1   // 味方の右下座標
) {
    // ショットと味方との当たり判定処理：
    // 当たったときにはショットを消し、
    // かわりに強いショットを出す。
    // 具体的な処理は、
    // DeleteShot、SpecialShotの各関数で行うとする。
    // ショットを消さずに、
    // ショットの強さだけを変える方法もある。
    if (cx0<sx1 && sx0<cx1 && cy0<sy1 && sy0<cy1) {
        DeleteShot();
        SpecialShot();
    }
}
```

自機の色を切り替えて弾を吸収する

自機の色を切り替えて、自機と同じ色になった弾を吸収するという攻撃方法です。「斑鳩（いかるが → P. 324）」に導入されたこの要素は、色の切り替えを行うことによって自機が置かれた状況の有利不利がまったく変わってしまうという点で、非常にユニークなものだといえます。

「斑鳩」では自機の色を白と黒に入れ替えることができます（Fig. 5-31）。自機と同じ色の弾にぶつかると、弾を消すと同時にエネルギーとして吸収します。逆に自機と異なる色の弾にぶつかると、自機は破壊されてしまいます。弾の状況を見定めて自機の色を入れ替えることがゲームの大きなポイントになっています。

Fig. 5-31 自機の色を切り替えて弾を吸収する

なお、「斑鳩」では敵やショットにも白と黒があり、撃ち込むショットの色を敵の色と逆にすると大ダメージを与えることができます。ショットの色は自機の色と連動して変わるので、綿密な戦術の研究が必要なゲームにしあがっています。

弾にぶつかったときの自機の色による効果の違いをまとめてみました（Fig. 5-32）。簡単に表現すれば次のようになります。

- **自機と弾の色が同じならば吸収**
- **自機と弾の色が異なればミス**

自機と弾の色を比べた結果によって処理を分岐させればよいのです。プログラムにまとめるとList 5-15のようになります。

サンプル

自機の色を切り替えて弾を吸収する → P. 319

Fig. 5-32 自機の色による効果の違い

	自機が白い状態	自機が黒い状態
白い弾にぶつかる	吸収 （弾を消し、エネルギーを増やす）	ミス
黒い弾にぶつかる	ミス	吸収 （弾を消し、エネルギーを増やす）

List 5-15 自機の色を切り替えて弾を吸収する

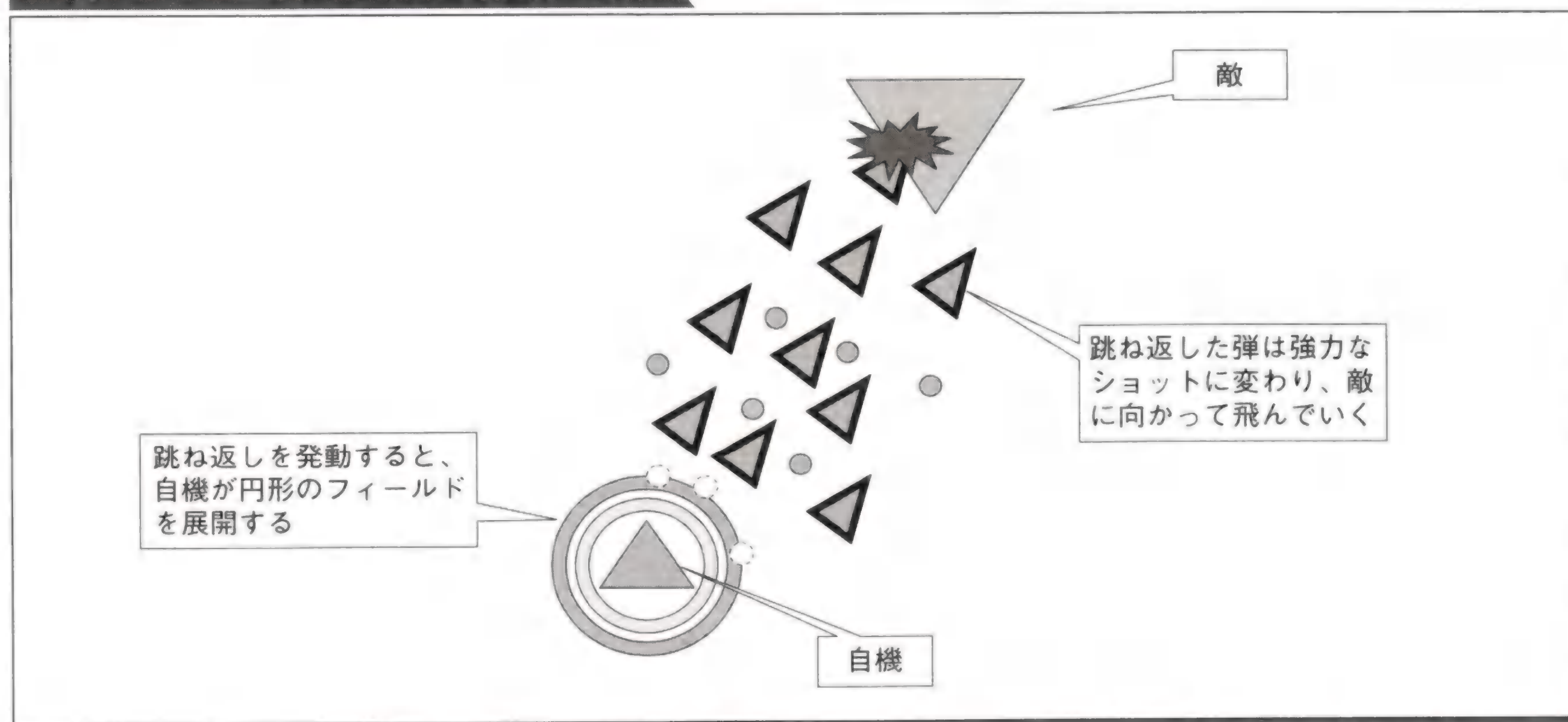
```
void SwitchColors(
    bool color_button,    // 色切り替えボタンの状態
    bool shot_button,     // ショットボタンの状態
    int num_bullet,       // 弾の数
    bool bullet_color[]   // 弾の色
) {
    static bool prev_col_button=false;  // 前回のボタンの状態
    static bool color=true;             // 自機の色
    static int energy=0;                // エネルギー

    // 色を切り替える
    if (!prev_col_button && color_button) color=~color;
    prev_col_button=color_button;

    // ショットを撃つ：
    // 自機と同じ色のショットを撃つ。
    // ショットの具体的な処理はShot関数で行うとする。
    if (shot_button) Shot(color);

    // 弾との当たり判定処理：
    // 判定の具体的な処理はHitBullet関数で行うとする。
    for (int i=0; i<num_bullet; i++) {
        if (HitBullet(i)) {

            // 同じ色ならば吸収：
            // 弾を消し、エネルギーを増やす。
            // 消去の具体的な処理はDeleteBullet関数で行うとする。
            if (color==bullet_color[i]) {
                DeleteBullet(i);
                energy++;
            }
```

```
            // 異なる色ならばミス：
            // ミスの具体的な処理はMiss関数で行うとする。
            else {
                Miss();
            }
        }
    }
}
```

敵弾をショットとして跳ね返す

自機で敵弾に体当たりして跳ね返し、逆にショットとして敵に撃ち込むというものです。この攻撃方法は「ギガウィング（→ P. 325）」シリーズに見られます。よけきれないような高密度の弾幕を逆に強力な反撃手段として変えてしまうという、ユニークな攻撃です。

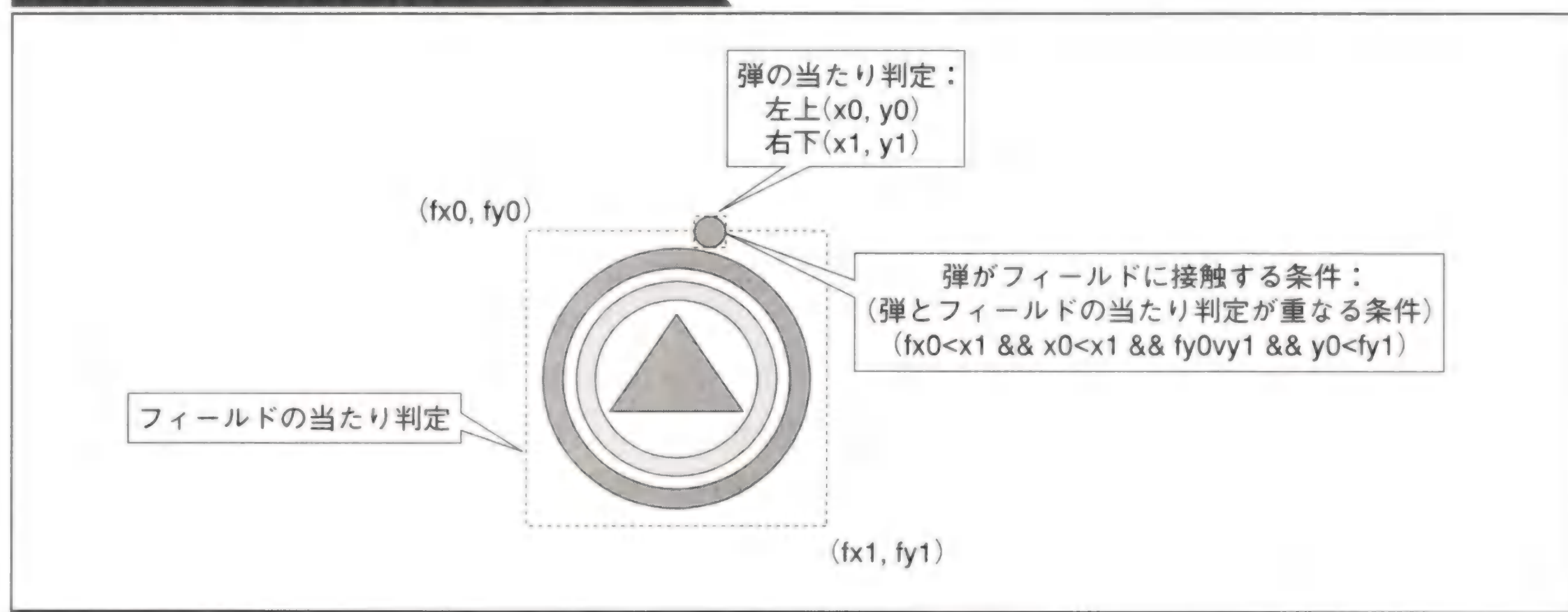
「ギガウィング」シリーズの場合、跳ね返し（「リフレクトフォース」と呼ばれる）を発動すると自機の周囲に円形のフィールドが発生し、このフィールドを敵弾にぶつけると強力なショットに変わって、自動的に敵に向かって飛んでいきます（Fig. 5-33）。普通のゲームでは弾幕をよけますが、この跳ね返し攻撃があると、むしろ高密度の弾幕を狙って突入していくプレイスタイルになります。

Fig. 5-33 敵弾をショットとして跳ね返す

さすがに無制限に敵弾を跳ね返せるのではゲームにならないので、「ギガウィング」シリーズの場合にはゲージによってこの攻撃手段の使用頻度を制限しています。弾の跳ね返しを一度行うと、一定時間が経過してゲージが溜まるまで、次の跳ね返しは発動できなくなります。

弾をフィールドで跳ね返すには、弾とフィールドとの間で当たり判定処理を行います(Fig. 5-34)。弾の左上座標を(x0, y0)、右下座標を(x1, y1)、フィールドの左上座標を(fx0, fy0)、右下座標を(fx1, fy1)とすると、弾がフィールドに接触する条件は次のようになります。

(fx0<x1 && x0<fx1 && fy0<y1 && y0<fy1)

この式が成立したら弾がフィールドに接触したということなので、弾を消してかわりにショットを発射します。

Fig. 5-34 弾とフィールドの当たり判定処理

跳ね返し攻撃で発生したショットは、自動的に敵に向かって飛んでいきます。敵に向かって飛ぶショットは、「狙い撃ち弾」(→ P. 10)や「ロックショット」(→ P. 129)と同じ要領で作ることができます。なお、ショット向けてを撃つ敵の選び方としては、もっとも近い敵を選ぶ方法や、跳ね返した弾を撃った敵を選ぶ方法などがあります。弾を撃った敵を狙うには、弾の座標や速度などといっしょに「その弾をどの敵が撃ったのか」という情報を保存しておきます。

跳ね返し攻撃のプログラムをList 5-16にまとめました。このプログラムは敵に向かって撃つショットの速度を求める処理も含んでいます。また、ショットは弾を撃った敵を狙います。

サンプル

敵の弾をショットとして跳ね返す　→ P. 319

List 5-16 敵弾をショットとして跳ね返す

```
#include <math.h>

void ReflectBullet(
    float fx0, float fy0,     // フィールドの左上座標
    float fx1, float fy1,     // フィールドの右下座標
    int num_bullet,           // 弾の数
    float x0[], float y0[],   // 弾の左上座標
    float x1[], float y1[],   // 弾の右下座標
    float x[], float y[],     // 弾の中心座標
    float ex[], float ey[],   // 弾を撃った敵の中心座標
    float speed               // ショットの速さ
) {
    // すべての弾に関して処理を行う
    for (int i=0; i<num_bullet; i++) {

        // 弾とフィールドとの当たり判定処理：
        // 弾がフィールドに接触したら、
        // 弾を消して、同じ位置から敵に向かってショットを撃つ。
        if (fx0<x1[i] && x0[i]<fx1 &&
            fy0<y1[i] && y0[i]<fy1) {

            // 弾の消去：
            // 具体的な処理はDeleteBullet関数で行うとする。
            DeleteBullet(i);

            // ショットの発射：
            // 敵に向かってショットを撃つ。
            // 発射の具体的な処理はShot関数で行うとする。
            // なお、厳密にはdが0のときの処理が必要。
            float vx=ex[i]-x[i], vy=ey[i]-y[i];
            if (vx!=0 || vy!=0) {
                float d=sqrt(vx*vx+vy*vy);
                vx*=speed/d;
                vy*=speed/d;
                Shot(x[i], y[i], vx, vy);
            }
        }
    }
}
```

レーザー同士をぶつける

敵のレーザーに自機のレーザーを正面からぶつけることによって、より強力な攻撃を発生させ、敵に大ダメージを与えるという攻撃方法です(Fig. 5-35、5-36)。この攻撃方法は「Gダライアス(→ P. 328)」や「ボーダーダウン(→ P. 333)」などにあります。「Gダライアス」の場合には、敵のレーザーを押し返して、もっと太いレーザーを敵にあびせることができます。「ボーダーダウン」の場合には、ボムのような強力な攻撃が発生します。

レーザー同士がぶつかったことを判定するにはFig. 5-37のようにします。レーザーはいくつもの小さな部分から構成されているので、各部分同士で当たり判定処理を行います。レーザーを正面から撃ち合っているときには、主に先端部分同士で消し合う結果になるでしょう。

Fig. 5-35 敵のレーザーに自機のレーザーをぶつける

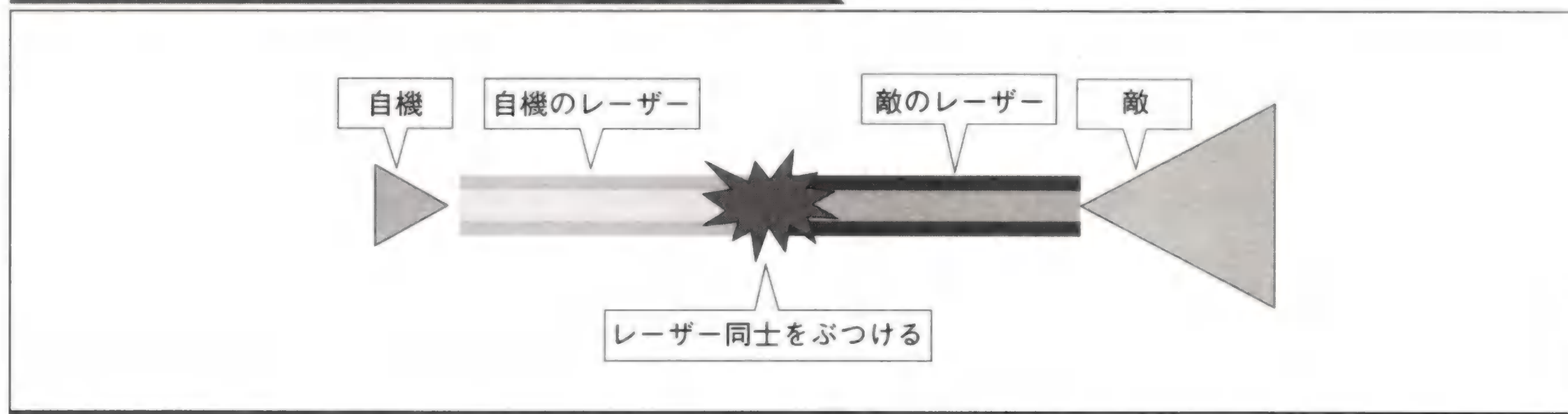



Fig. 5-36 強力な攻撃が発生する

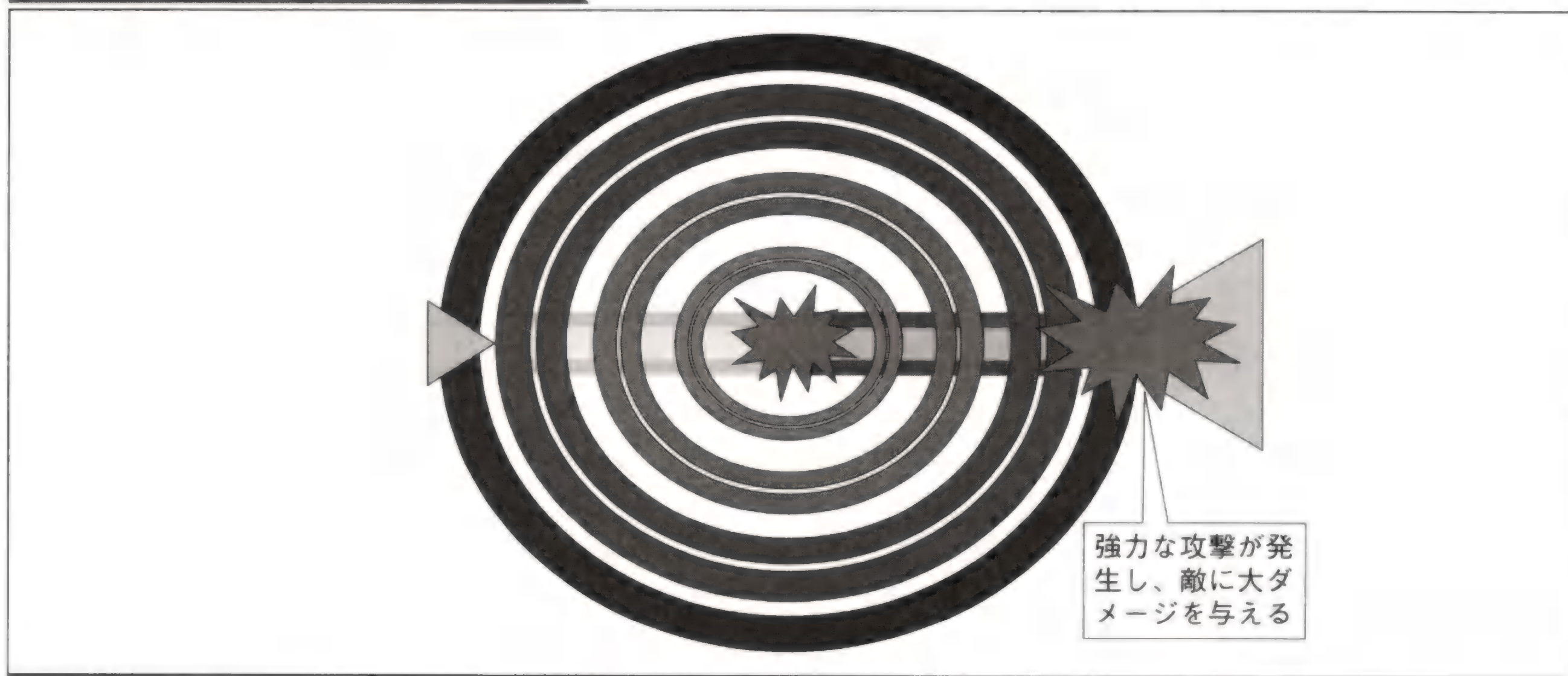



Fig. 5-37 レーザー同士の当たり判定処理

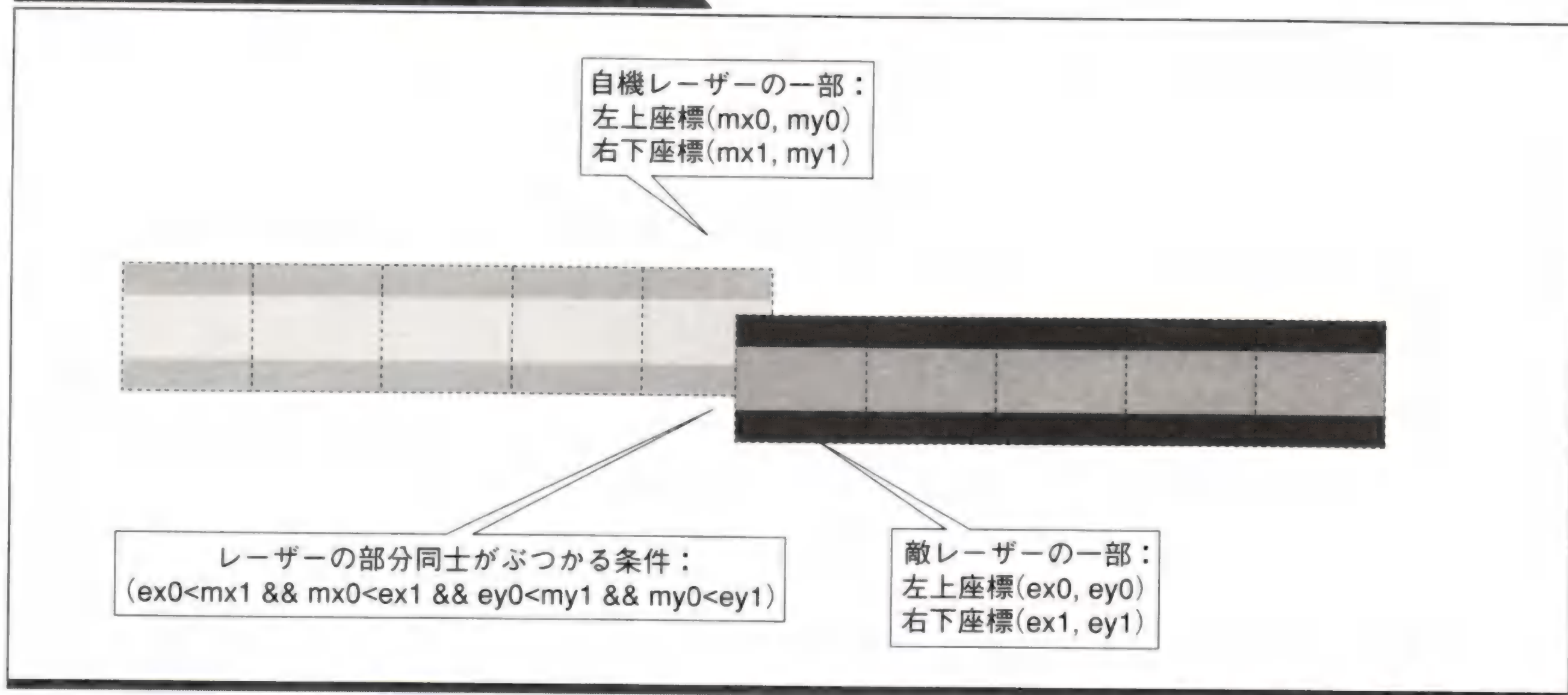



Fig. 5-37の場合、自機レーザーの一部と敵レーザーの一部がぶつかる条件は次のようになります。

(ex0<mx1 && mx0<ex1 && ey0<my1 && my0<ey1)

レーザーの部分同士がぶつかったら、ぶつかった2つの部分を消滅させて、攻撃を発生させるためのエネルギーを増やします。そしてエネルギーが一定値以上になったら、Fig. 5-36のような強力な攻撃を発生させます。

サンプル

レーザー同士をぶつける → P. 319

List 5-17 レーザー同士をぶつける

```
void LaserCollision(
    float mx0[], float my0[],  // 自機レーザーの一部の左上座標
    float mx1[], float my1[],  // 自機レーザーの一部の右下座標
    int num_my_lasers,         // 自機レーザーの部分の数
    float ex0[], float ey0[],  // 敵レーザーの一部の左上座標
    float ex1[], float ey1[],  // 敵レーザーの一部の右下座標
    int num_enemy_lasers       // 敵レーザーの部分の数
) {
    static int energy=0;         // エネルギー
    static int min_energy=100;   // 攻撃の発生に必要なエネルギー

    // レーザー同士の当たり判定処理：
    // 自機レーザーと敵レーザーの各部分同士がぶつかったら、
```

```
    // 2つの部分を消滅させてエネルギーを増やす。
    // 消滅の具体的な処理は、DeleteMyLaser、
    // DeleteEnemyLaserの各関数で行うとする。
    for (int i=0; i<num_my_lasers; i++) {
        for (int j=0; j<num_enemy_lasers; j++) {
            if (ex0[j]<mx1[i] && mx0[i]<ex1[j] &&
                ey0[j]<my1[i] && my0[i]<ey1[j]) {
                DeleteMyLaser(i);
                DeleteEnemyLaser(j);
                energy++;
            }
        }
    }

    // 攻撃の発生：
    // エネルギーが必要量に達していたら、
    // 強力な攻撃を発生させる。
    // 発生の具体的な処理はSpecialAttack関数で行うとする。
    if (energy>=min_energy) SpecialAttack();
}
```

敵弾に自機をかすらせてパワーアップする

敵弾に自機をかすらせることによって、自機をパワーアップさせたりスコアを得たりする攻撃方法です (Fig. 5-38)。この攻撃方法は「サイヴァリア (→ P. 327)」で導入され、「式神の城 (→ P. 327)」などにも採用されています。「サイヴァリア」では敵弾に自機をかすらせることを「BUZZ (バズ)」と呼び、このBUZZを自機パワーアップやスコア稼ぎといったゲーム性の中軸としています。「式神の城」では敵弾に自機をかすらせることによって「テンションゲージ」が増加し、ボーナスの倍率が上がったり、一時的に強いショットが撃てるようになったりします。

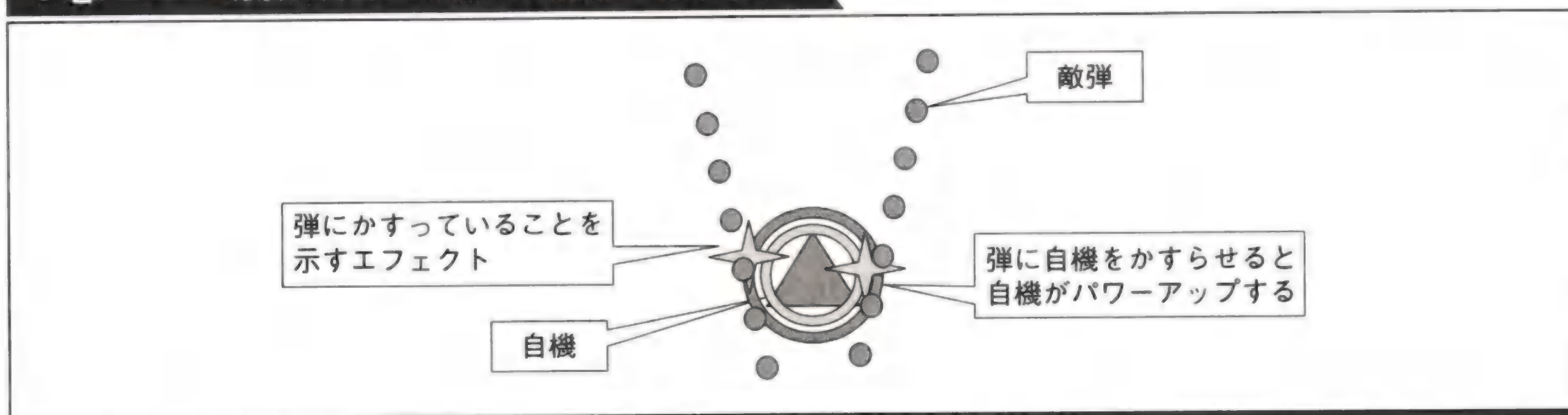
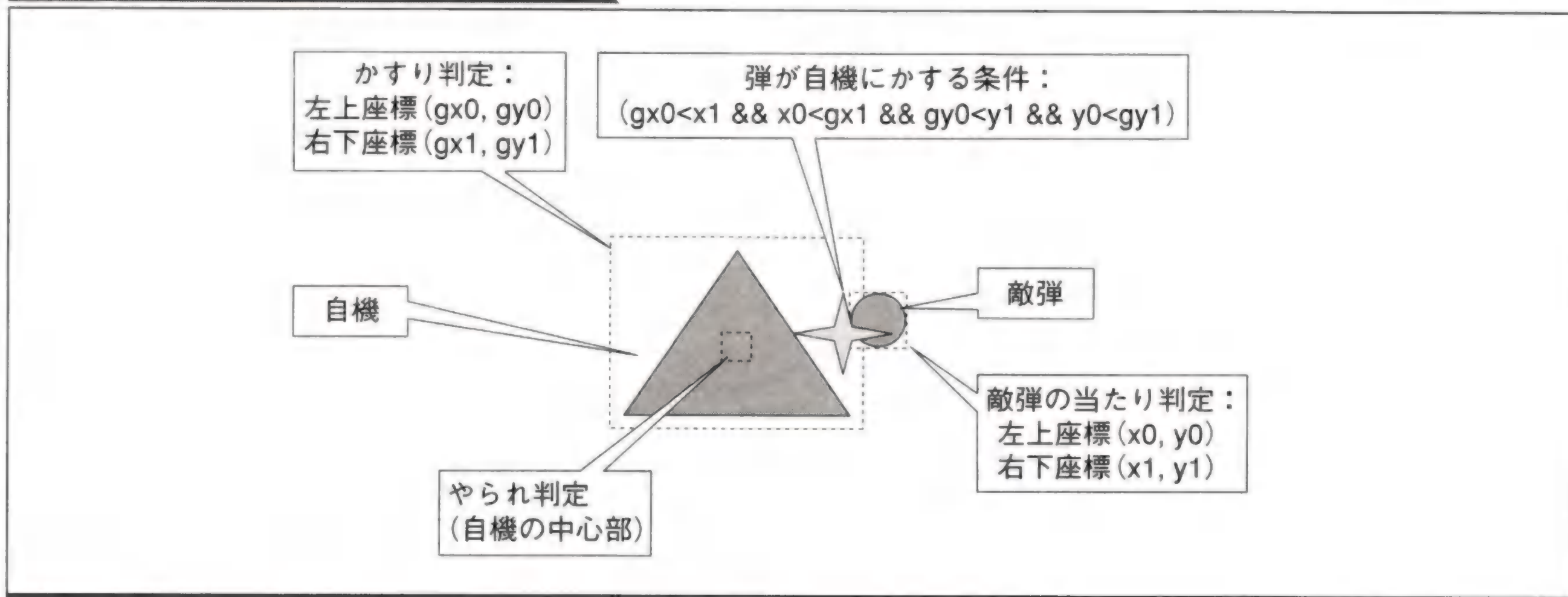
Fig. 5-38 敵弾に自機をかすらせてパワーアップする

「敵弾に自機をかすらせる」というのは、「自機を敵弾にやられないギリギリの距離まで近づける」ということです。そのためには、自機に「やられ判定」と「かすり判定」という2つの当たり判定を用意しておきます(Fig. 5-39)。やられ判定は自機の中心に配置し、かすり判定は自機と同じかやや大きめくらいに設定します。ちょうど、やられ判定の周囲をかすり判定が覆う形になります。

Fig. 5-39 やられ判定とかすり判定

弾の当たり判定の左上座標を(x0, y0)、右下座標を(x1, y1)、かすり判定の左上座標を(gx0, gy0)、右下座標を(gx1, gy1)とすると、弾が自機をかすす条件は次のようになります。

(gx0<x1 && x0<gx1 && gy0<y1 && y0<gy1)

これは弾の当たり判定とかすり判定とが重なる条件です。

■ 同じ弾に何度かすれるか

同じ弾に自機が何度もかすれるかどうかはゲームによります。「サイヴァリア」では1つの弾には一度しかかすることができませんが、「サイヴァリア リビジョン(→ P. 327)」や「式神の城」では同じ弾に何度もかすることができます。

1つの弾に一度しかかすれないゲームの場合には、弾に「通常状態」と「かすり状態」を用意しておいて、弾が自機に一度かすったらかすり状態に移行するようにします(Fig. 5-40)。そして一度かすり状態になったら、もう自機とはかすらないようにします。

1つの弾に何度もかすれるゲームの場合にも、弾に通常状態とかすり状態を用意するのは同じですが、一定時間が経ったらかすり状態から通常状態に戻るようにします(Fig. 5-41)。そして通常状態に戻ったら、再びかすることができるようにします。

1つの弾に何度もかすれる場合について、弾を自機にかすらせる処理をまとめたのがList 5-18です。後半の処理をなくせば、1つの弾に一度しかかすれなくなります。

サンプル

敵弾に自機をかすらせる → P. 319

Fig. 5-40 一度しかかすれない場合の状態遷移

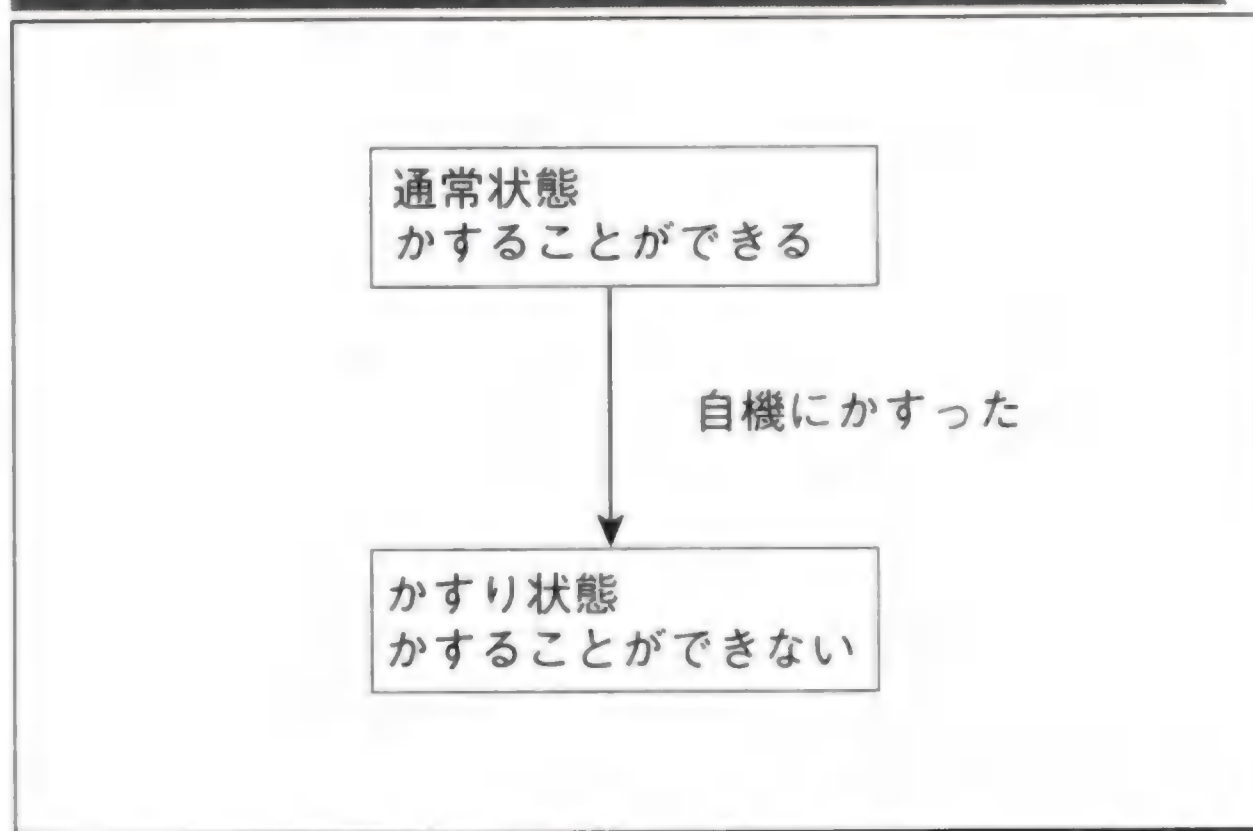
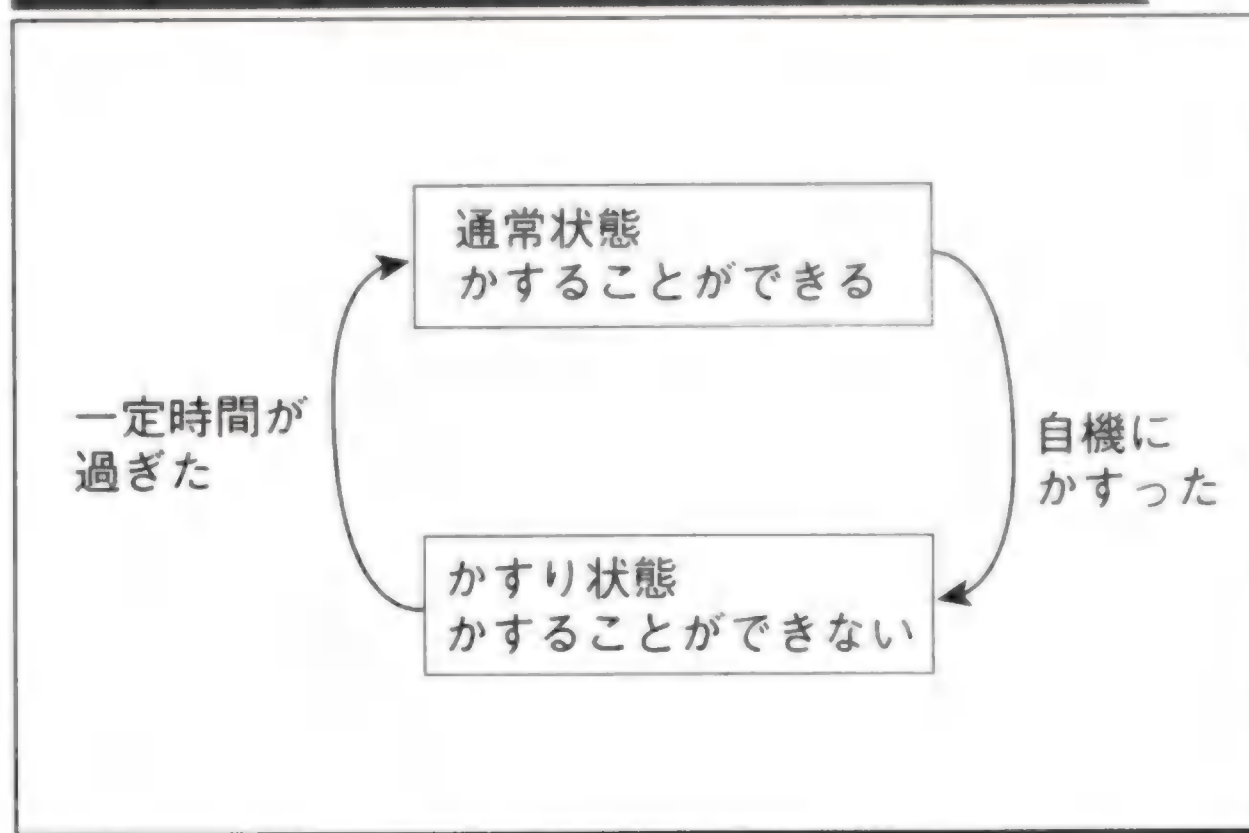



Fig. 5-41 何度もかすれる場合の状態遷移

List 5-18 敵弾に自機をかすらせてパワーアップする

```
void GrazeBullet(
    float gx0, float gy0,     // かすり判定の左上座標
    float gx1, float gy1,     // かすり判定の右下座標
    int num_bullet,           // 弾の数
    float x0[], float y0[],   // 弾の当たり判定の左上座標
    float x1[], float y1[],   // 弾の当たり判定の右下座標
    bool grazing[],           // 弾がかすっているかどうか
    int time[],               // かすり時間
    int& power                // 自機のパワー
                              // （経験値、得点倍率など）
) {
    // すべての弾について処理を行う
    for (int i=0; i<num_bullet; i++) {

        // 通常状態：
        // 当たり判定処理を行い、弾が自機にかすったら、
        // かすり状態にする。
        // かすり状態の残り時間を設定し、自機のパワーを増やす。
        if (!grazing[i]) {
            if (gx0<x1[i] && x0[i]<gx1 &&
                gy0<y1[i] && y0[i]<gy1) {
                grazing[i]=true;
                time[i]=20;
                power++;
            }
        }
```

```
        // かすり状態：
        // かすり状態の残り時間を減らし、
        // 時間が0になったら通常状態に戻る。
        // この処理をなくせば、1つの弾に一度しかかすれなくなる。
        else {
            if (time[i]==0) grazing[i]=false; else time[i]--;
        }
    }

}
```

弾や敵の動きをスローにする

弾や敵の動きを一定期間だけ遅くして、弾よけをしやすくするという攻撃&防御の方法です(Fig. 5-42、5-43)。この方法は「エスプガルーダ(→ P. 324)」に見られます。「エスプガルーダ」では、ボタンを押している間だけ弾や敵の動きを遅くすることができます。ボタンの使用期間はゲージで制限されていて、ゲージが0になるとペナルティとして弾の動きが速くなります(Fig. 5-44)。

この攻撃方法が面白いのは、弾や敵の移動速度が落ちるだけではなく、処理落ちしたように弾や敵の行動速度が落ちるということです。これはスロー状態のときに弾や敵の行動回数を半分にしてやれば、ちょうど処理落ちしたのと同じ状態を再現することができます。

弾や敵の動きをスローにする処理をまとめたものがList 5-19です。スロー状態のときの弾や敵の行動回数を半分にしましたが、timeの値を変えればもっと遅くすることもできます。

Fig. 5-42 通常状態の弾

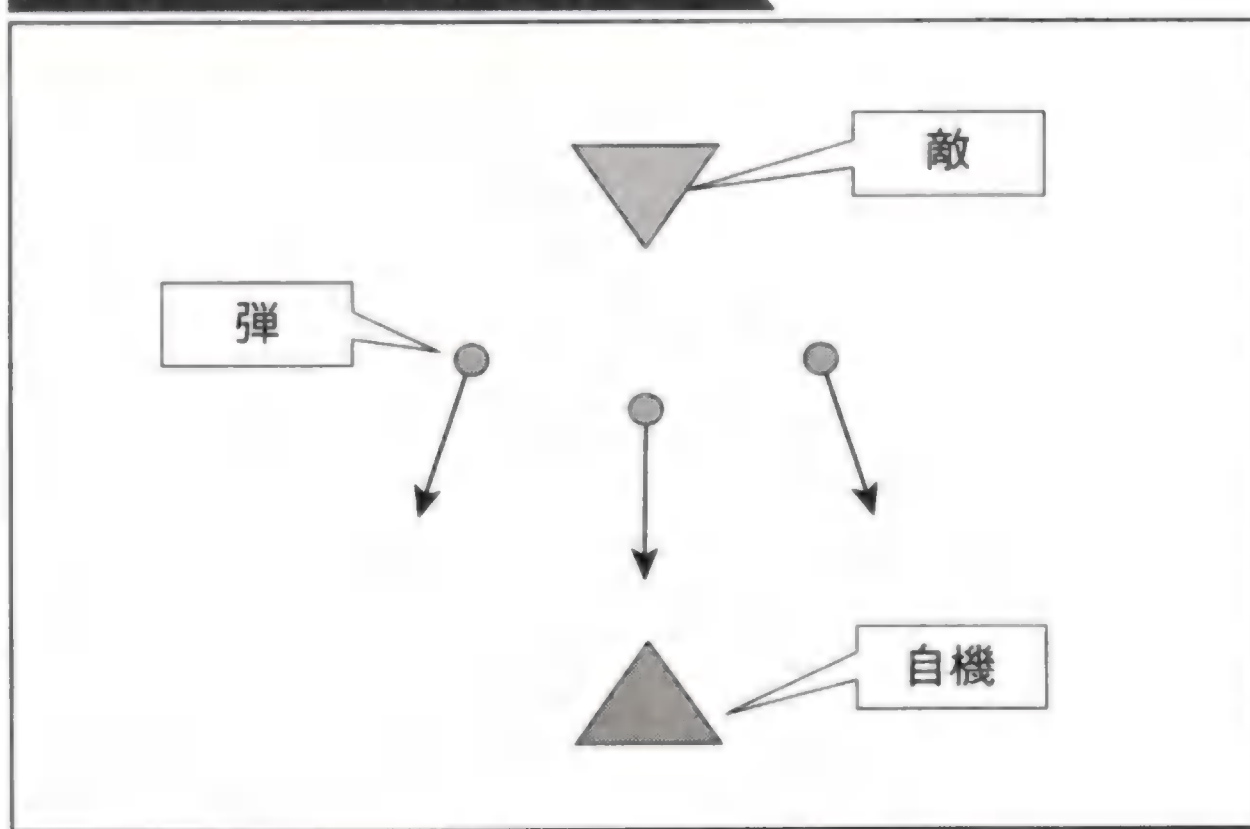



Fig. 5-43 弾の動きを遅くする

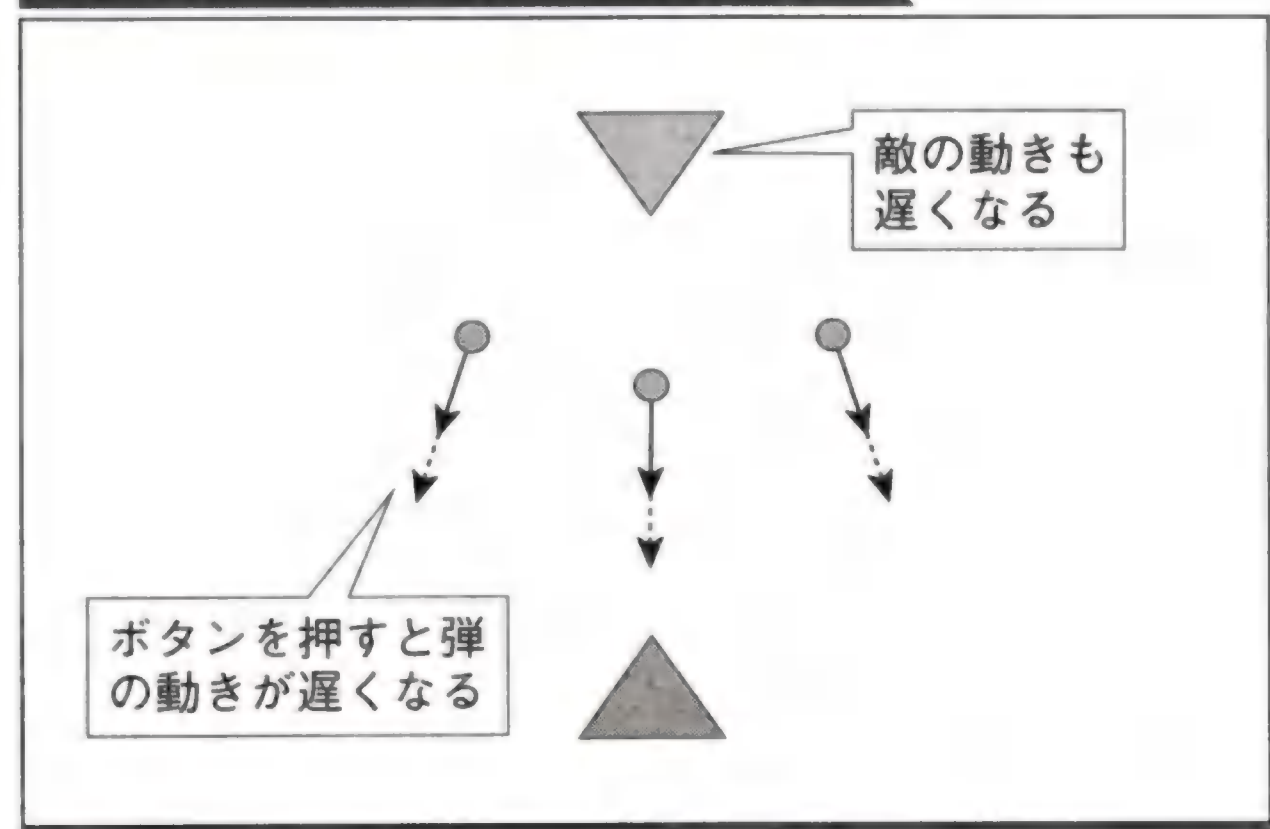
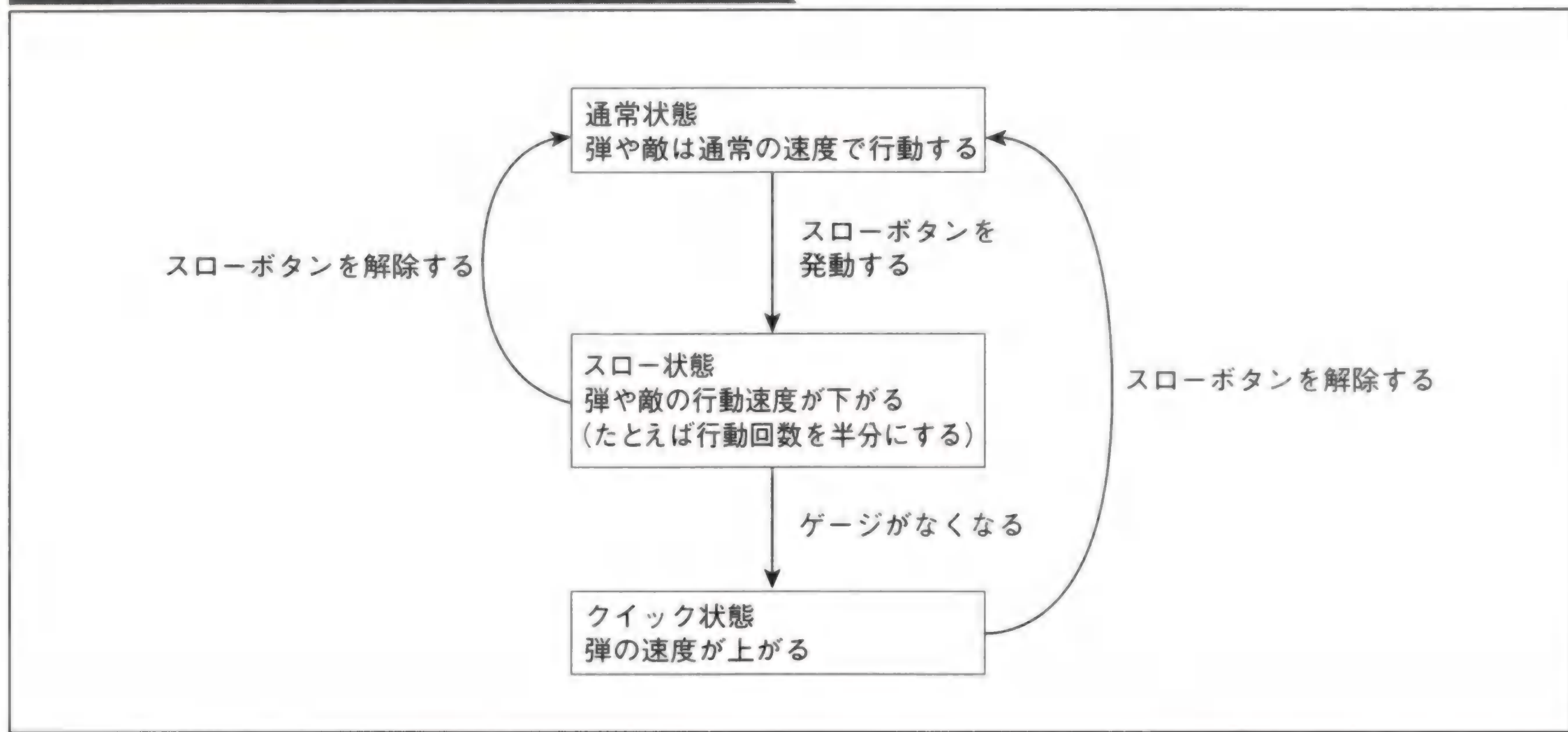



Fig. 5-44 「エスプガルーダ」の場合の状態遷移

サンプル

弾や敵の動きをスローにする → P. 319

List 5-19 弾や敵の動きをスローにする

```
void Slowing(
    int& gauge,  // スロー状態にするために必要なゲージ
    bool button  // スローボタンを押しているかどうか
) {
    // タイマー：
    // スロー状態のときに行動回数を調整するために使う。
    static int time=1;

    // 敵と弾を動かす：
    // 通常状態のときには毎回動かし、
    // スロー状態のときには2回に1回だけ動かす。
    // 移動の具体的な処理は、
    // MoveEnemy、MoveBulletの各関数で行うとする。
    if (!button || (gauge>0 && time==0)) {
        MoveEnemy();
        MoveBullet();
    }

    // ペナルティ：
    // ゲージが0なのにスローボタンを押していたら、
    // ペナルティとして弾をもう一度動かす。
```

```
    // 結果として、弾は通常の2倍の速度で動く。
    if (button && gauge==0) {
        MoveBullet();
    }

    // タイマーの更新
    if (time>0) time--; else time=1;
}
```

自由に動かせる照準

ボタンを押している間は照準を自由に動かすことができ、ボタンを離すと照準の位置に強力な攻撃が発生するという攻撃方法です(Fig. 5-45、5-46)。この攻撃方法は「式神の城(→ P. 327)」シリーズに見られます。照準を動かしている間は自機を動かせなくなるので、いかに敵の隙をついて照準を合わせるかがゲームのポイントになります。

この攻撃方法には3つの状態があります (Fig. 5-47)。通常状態ではスティック入力で自機が動き、照準状態 (照準が出ている状態) では照準が動きます。照準ボタンを離すと攻撃状態 (照準の位置に攻撃が出ている状態) となり、自機は動かせるようになりますが、一定時間が経過して攻撃が終わるまでは次の照準を出すことはできません。

自由に動かせる照準に関するプログラムはList 5-20のようになります。

Fig. 5-45 自由に動かせる照準

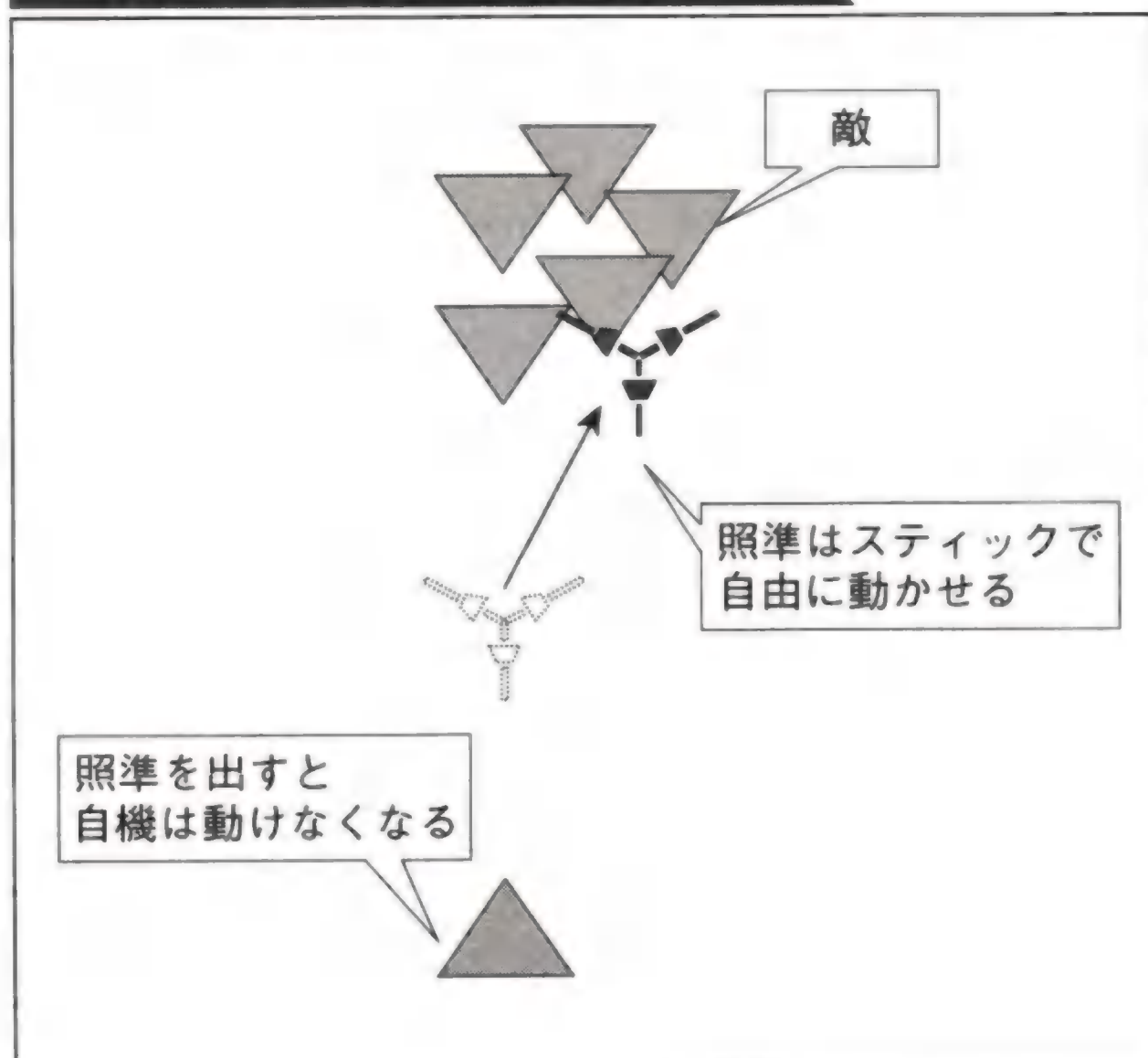



Fig. 5-46 照準を使った攻撃

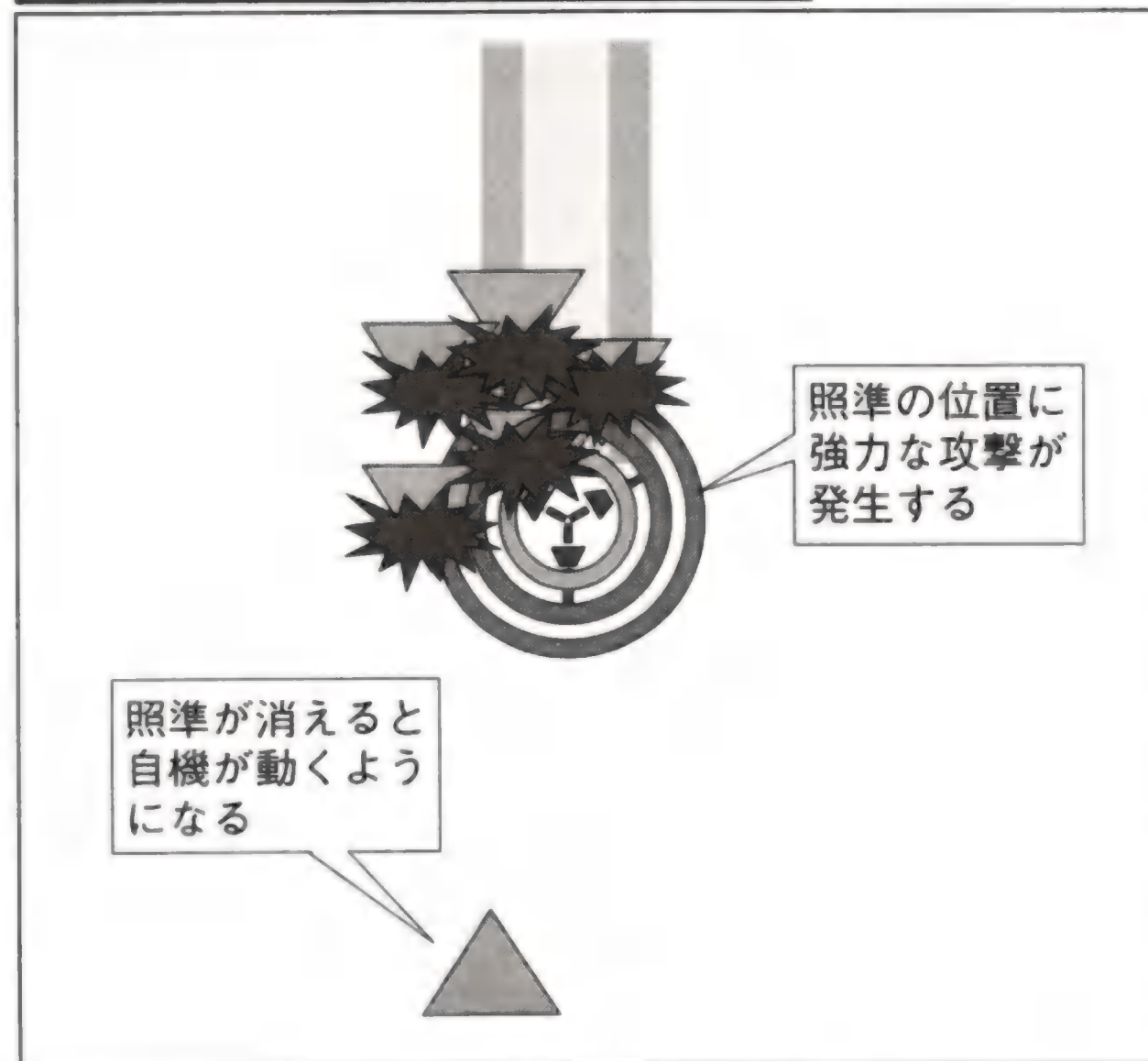
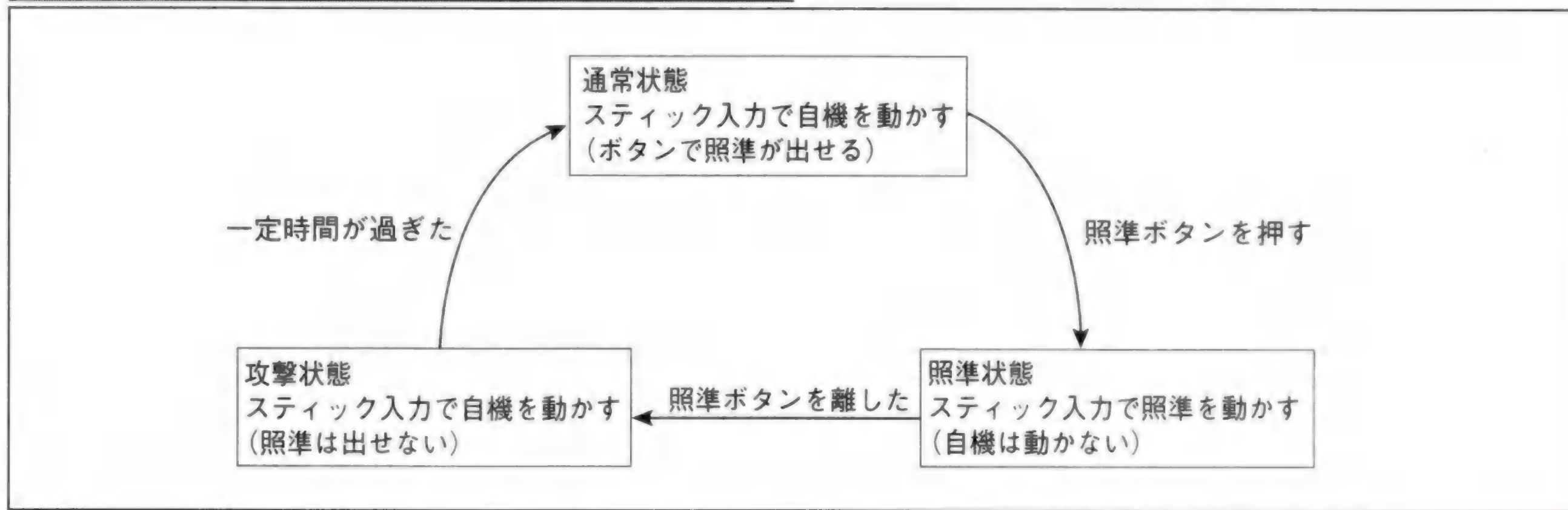



Fig. 5-47 自由に動かせる照準に関する状態遷移

サンプル

自由に動かせる照準 → P. 320

List 5-20 自由に動かせる照準

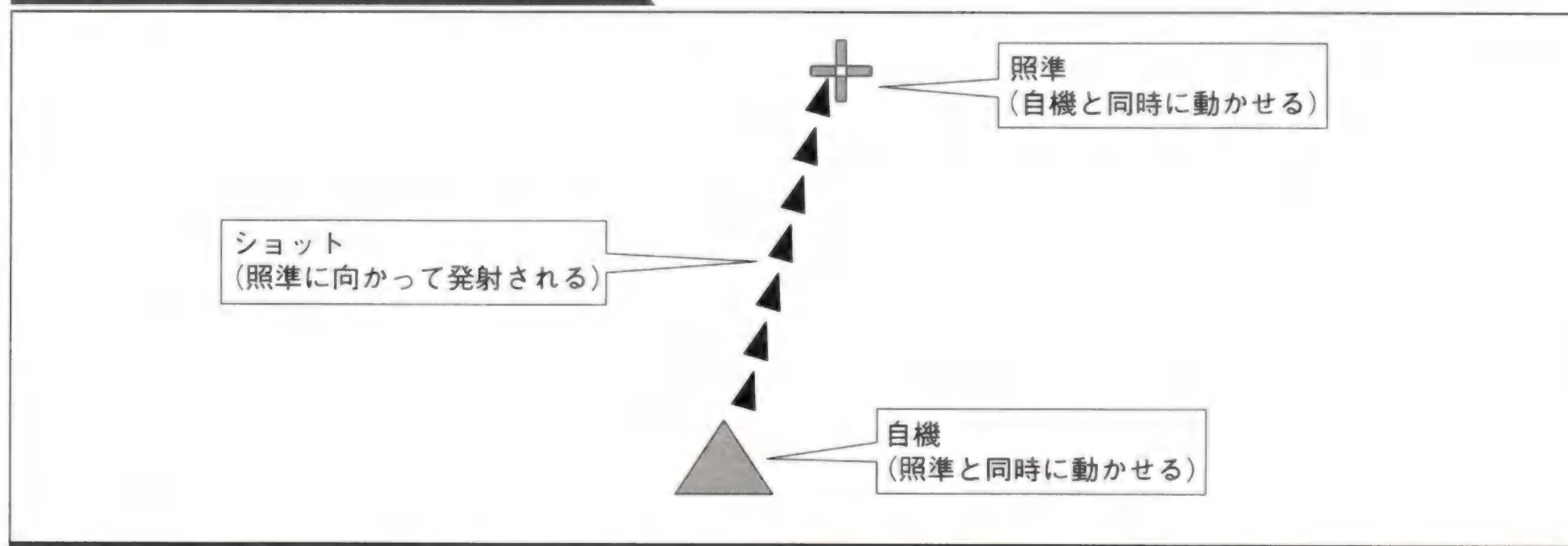
```
// 状態（通常、照準、攻撃）
typedef enum {
    NORMAL, SIGHT, ATTACK
} STATE_TYPE;

// 自機と照準を動かす
void MoveSight1(
    float& x, float& y,      // 自機の座標
    float& sx, float& sy,    // 照準の座標
    float speed,             // 自機の移動の速さ
    float sight_speed,       // 照準の移動の速さ
    bool up, bool down,      // スティック入力（上下）
    bool left, bool right,   // スティック入力（左右）
    bool button              // 照準ボタン入力
) {
    static int state=NORMAL;  // 状態（通常から開始）
    static int time;          // 攻撃時間

    // 状態に応じて分岐する
    switch (state) {

        // 通常状態：
        // 自機を移動し、
        // ボタンが押されたら照準を出して、照準状態に移る。
        case NORMAL:
            if (up   ) y-=speed;
```

```
        if (down ) y+=speed;
        if (left ) x-=speed;
        if (right) x+=speed;
        if (button) {
            state=SIGHT;
            sx=x; sy=y;
        }
        break;

    // 照準状態:
    // 照準を移動し、ボタンが離されたら攻撃状態に移る。
    case SIGHT:
        if (up   ) sy-=sight_speed;
        if (down ) sy+=sight_speed;
        if (left ) sx-=sight_speed;
        if (right) sx+=sight_speed;
        if (!button) {
            state=ATTACK;
            time=100;
        }
        break;

    // 攻撃状態:
    // 自機を移動し、一定時間が経ったら通常状態に戻る。
    // 攻撃の具体的な処理はAttack関数で行うとする。
    case ATTACK:
        Attack();
        if (up   ) y-=speed;
        if (down ) y+=speed;
        if (left ) x-=speed;
        if (right) x+=speed;
        if (time==0) state=NORMAL;
        time--;
    break;
  }
}
```

自機と照準を同時に動かす

自機を動かして敵や弾をよけつつ、同時に照準も操作して攻撃を行うという攻撃方法です（Fig. 5-48）。自機のショットは照準に向かって発射されます。この独特な攻撃方法は「SDI（→P. 324）」に見られます。「SDI」は非常にユニークな操作系のゲームで、自機はスティックで動かし、照準はトラックボールで動かします。いってみれば自機が動かせる「ミサイルコマンド（→ P. 333）」といったところでしょうか。スティックの上部にショットボタンがついているという点でもほかに類を見ないゲームです。

自機と照準を同時に動かすには、自機用のスティック入力を基に自機を動かし、照準用のスティック入力（あるいはトラックボール入力）を基に照準を動かすだけです。自機から照準に向かうショットは、「狙い撃ち弾」（→ P. 10）や「ロックショット」（→ P. 129）と同じ要領で実現できます。List 5-21はこれらの処理をまとめたプログラムです。

Fig. 5-48 自機と照準を同時に動かす

サンプル

自機と照準を同時に動かす　→ P. 320

List 5-21 自機と照準を同時に動かす

```
#include <math.h>

void MoveSight2(
    float& x, float& y,        // 自機の座標
    float speed,               // 自機の速さ
    bool up1, bool down1,      // 自機用スティック入力1（上下）
    bool left1, bool right1,   // 自機用スティック入力1（左右）
    float& sx, float& sy,      // 照準の座標
    float sight_speed,         // 照準の速さ
    bool up2, bool down2,      // 照準用スティック入力2（上下）
    bool left2, bool right2,   // 照準用スティック入力2（左右）
    bool button,               // ショットボタンの入力
    float shot_speed           // ショットの速さ
) {
    // 自機を動かす
    if (up1    ) y-=speed;
    if (down1  ) y+=speed;
    if (left1  ) x-=speed;
    if (right1) x+=speed;

    // 照準を動かす
    if (up2    ) sy-=sight_speed;
    if (down2  ) sy+=sight_speed;
    if (left2  ) sx-=sight_speed;
    if (right2) sx+=sight_speed;

    // ショットを撃つ：
    // ショットボタンが押されたら、
    // 自機から照準に向かってショットを撃つ。
    // 発射の具体的な処理はShot関数で行うとする。
    // なお、厳密にはdが0のときの処理が必要。
    if (button) {
        float vx=sx-x, vy=sy-y;
        if (vx!=0 || vy!=0) {
            float d=sqrt(vx*vx+vy*vy);
            vx*=shot_speed/d;
            vy*=shot_speed/d;
            Shot(x, y, vx, vy);
        }
    }
}
```

Stage 5 のまとめ ▶▶

特殊攻撃はゲームを味わい深くするスパイスだともいえます。何か1つでも印象的な攻撃手段があるゲームは、それだけで強くプレイヤーに対して存在感をアピールすることができます。もちろん、特殊攻撃をたくさん盛り込めばよいというわけではなくて、そこは微妙なサジ加減が必要なのですが、ゲームをせっかく作るからには「ゲームの顔」になる要素を入れたいものです。

ということで、「特殊攻撃はゲームのスパイス、ただし入れすぎには注意！」というのが本章のまとめです。

Stage 06 ▶▶

敵 Enemy

シューティングゲームには「弾」に加えて「敵」も出現するのが普通です。弾はよけるだけですが、敵は自機からの攻撃で破壊することができます。敵はプレイヤーに攻めの楽しみを味わわせる要素だといえます。

敵はゲームの雰囲気を演出するうえでも重要な要素です。たとえば、宇宙を舞台にしたゲームならば宇宙船、空中ならば戦闘機、メルヘンチックなゲームならば動物やお菓子や果物といったように、多くのゲームは世界観に合わせた敵を用意しています。弾に比べて敵は大きいので、凝ったグラフィックスやアニメーションを使って、プレイヤーにゲームの雰囲気を伝えることができます。

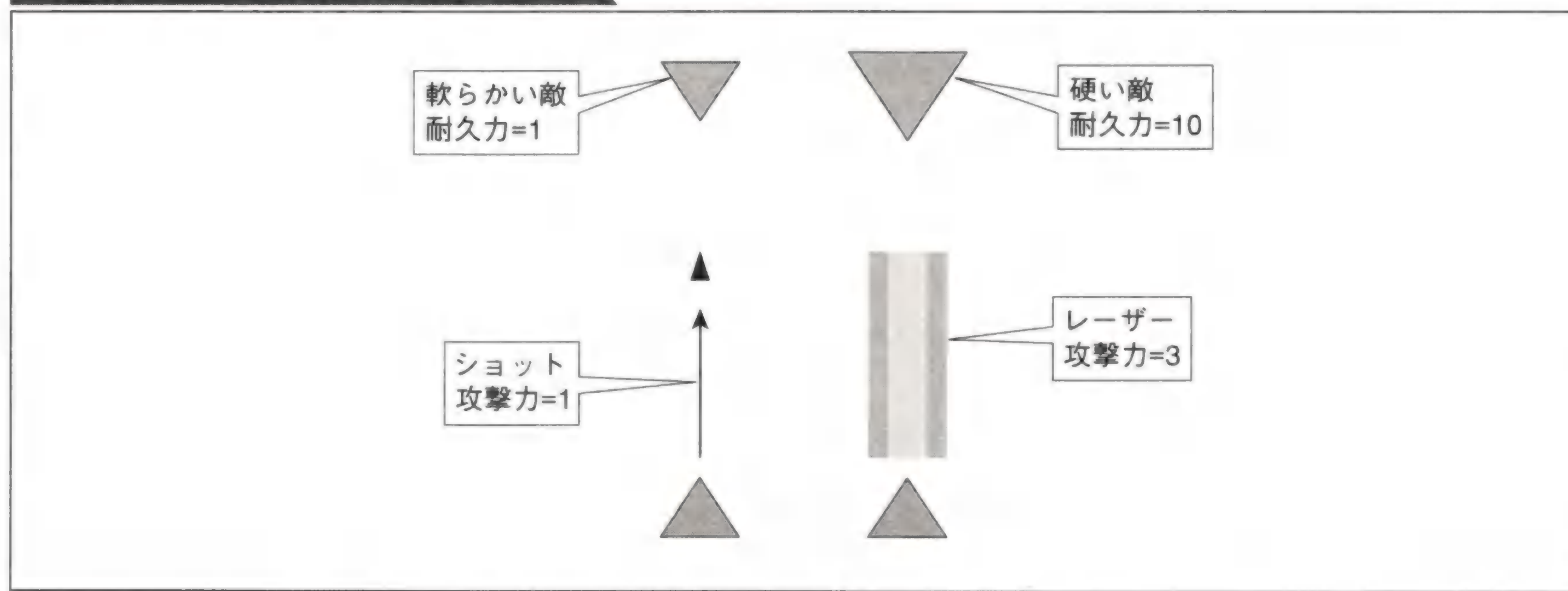
敵の動きに関しては、弾を動かす方法を応用することができます。動くものが弾か敵かという違いだけで、動きのアルゴリズムは敵と弾とで共通のことが多いのです。

本章では、弾と共通ではない、敵に特有のアルゴリズムを中心に解説します。最初は敵に関する基本的な事柄を説明し、続いて「編隊」「ボスキャラ」「触手」「多関節」といった応用的なトピックに進みます。

破壊できる敵

破壊できないことが多い弾とは違って、ほとんどの敵は破壊できます。ボスキャラなどでは中核となるパーツしか壊せなかったり、複数のパーツを特定の順番で壊さなければならないこともありますが、いずれにしても破壊することが可能です。

破壊できる敵には、ショットを1発当てるだけで壊せるものもあれば、レーザーでじっくり焼かないと壊せないものもあります。しかし「敵の耐久力」と「武器の攻撃力」という概念を使えば、どちらの敵も同じ方法で扱うことができます(Fig. 6-1)。

Fig. 6-1 敵の耐久力と武器の攻撃力

ショットを1発当てただけで壊せるような軟らかい敵は耐久力を低くし(たとえば1)、ショットを何発も当てないと壊せないような硬い敵は耐久力を高くします(たとえば10)。武器については、ショットのように弱い武器は攻撃力を低く(たとえば1)、レーザーのように強い武器は攻撃力を高くします(たとえば3)。

武器が敵に命中したら、敵の耐久力から武器の攻撃力を減算します(Fig. 6-2)。敵の耐久力をvit、武器の攻撃力をstrとすると次のように計算できます。

```
vit-=str
```

なお、武器が敵に命中する条件は次のとおりです。

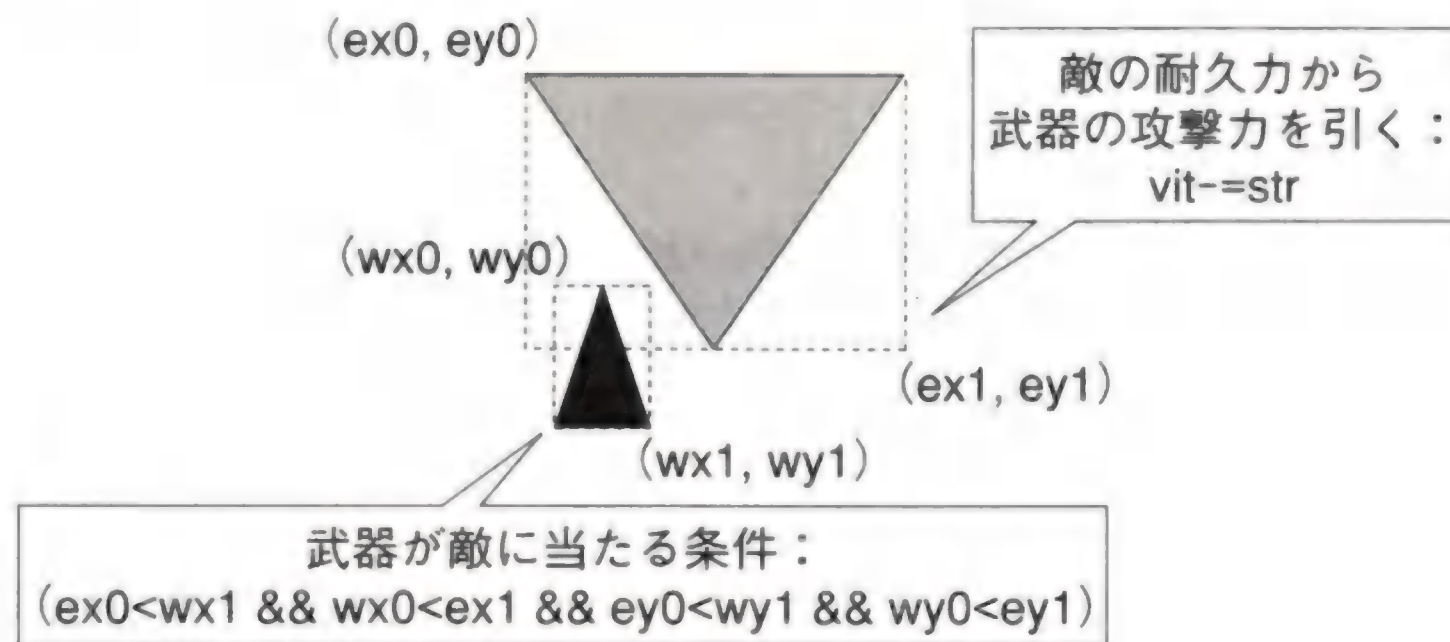
```
(ex0<wx1 && wx0<ex1 && ey0<wy1 && wy0<ey1)
```

耐久力が0またはマイナスになったら、その敵は破壊されたということです。この場合は敵を消滅させて、かわりに爆発などを表示します。

破壊できる敵に関する処理をまとめるとList 6-1のようになります。当たり判定処理を行い、当たったら武器の攻撃力で敵の耐久力を削ります。

サンプル

破壊できる敵 → P. 320

Fig. 6-2 武器が敵に命中したときの処理

List 6-1 破壊できる敵の処理

```
void BreakableEnemy(
    int num_enemy,                // 敵の数
    float ex0[], float ey0[],     // 敵の当たり判定の左上座標
    float ex1[], float ey1[],     // 敵の当たり判定の右下座標
    float vit[],                  // 敵の耐久力
    int num_weapon,               // 武器の数
    float wx0[], float wy0[],     // 武器の当たり判定の左上座標
    float wx1[], float wy1[],     // 武器の当たり判定の右下座標
    float str[]                   // 武器の攻撃力
) {
    // 敵と武器の当たり判定処理：
    // すべての敵と武器の組み合わせについて、
    // 武器が敵に当たったかどうかを判定する
    for (int i=0; i<num_enemy; i++) {
        for (int j=0; j<num_weapon; j++) {
            if (ex0[i]<wx1[j] && wx0[j]<ex1[i] &&
                ey0[i]<wy1[j] && wy0[j]<ey1[i]
            ) {
                // 当たった場合：
                // 武器の攻撃力で敵の耐久力を削り、
                // 耐久力が0または負になったら敵を破壊する。
                // 破壊の具体的な処理は、
                // DeleteEnemy関数で行うとする。
                vit[i]-=str[j];
                if (vit[i]<=0) DeleteEnemy(i);

                // 敵に当たった武器は消す：
```

```
                // 具体的な処理はDeleteWeapon関数で行うとする。
                DeleteWeapon(j);
            }
        }
    }
}
```

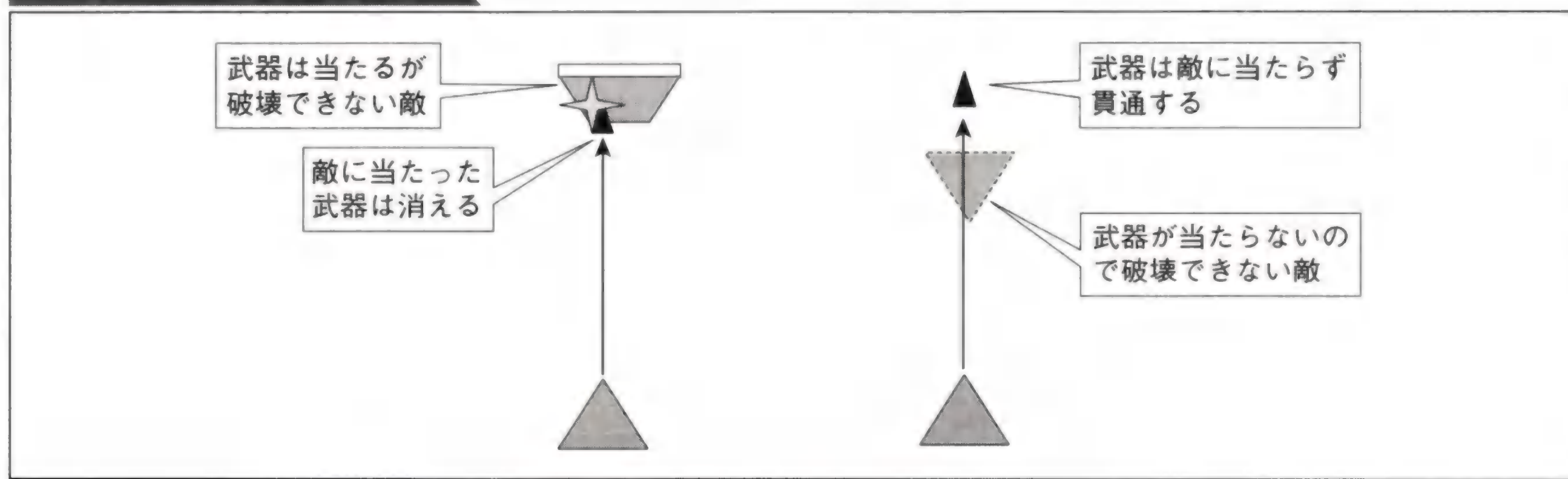
破壊できない敵

破壊できない敵には2種類あります（Fig. 6-3）。1つは武器が当たるけれども破壊できない敵で、もう1つは武器が当たらずに通り抜けてしまう敵です。1つ目の有名な例は「ゼビウス（→P. 329）」に出てくる「バキュラ」です。2つ目の例としては、一時的に透明になる敵などがあります。

破壊できない敵を扱うには、敵のタイプを記録しておき、タイプによって処理を分岐します。破壊できる敵も含めると次の3種類にタイプを分けます。

- **破壊できる敵**
- **武器は当たるが破壊できない敵**
- **武器が当たらず破壊できない敵**

この処理をまとめたのがList 6-2です。敵の種類が増えた場合にも、基本的な方法は変わりません。

Fig. 6-3 破壊できない敵

サンプル

破壊できない敵　→ P. 320

List 6-2 破壊できないものも含めた敵の処理

```
// 敵のタイプを表す定数:
// 破壊できる、破壊できない、武器が当たらない
typedef enum {
    BREAKABLE, UNBREAKABLE, TRANSPARENT
} ENEMY_TYPE;

// 破壊できないものも含めた敵の処理
void BreakableEnemy2(
    int num_enemy,                // 敵の数
    ENEMY_TYPE type[],            // 敵のタイプ
    float ex0[], float ey0[],     // 敵の当たり判定の左上座標
    float ex1[], float ey1[],     // 敵の当たり判定の右下座標
    float vit[],                  // 敵の耐久力
    int num_weapon,               // 武器の数
    float wx0[], float wy0[],     // 武器の当たり判定の左上座標
    float wx1[], float wy1[],     // 武器の当たり判定の右下座標
    float str[]                   // 武器の攻撃力
) {
    // すべての敵に関するループ
    for (int i=0; i<num_enemy; i++) {

        // 武器に当たらない敵の場合:
        // 当たり判定処理は行わず、次の敵の処理に移る。
        if (type[i]==TRANSPARENT) continue;

        // 敵と武器との当たり判定処理
        for (int j=0; j<num_weapon; j++) {
            if (ex0[i]<wx1[j] && wx0[j]<ex1[i] &&
                ey0[i]<wy1[j] && wy0[j]<ey1[i]
            ) {
                // 破壊できる敵に当たった場合:
                // 武器の攻撃力で敵の耐久力を削り、
                // 耐久力が0または負になったら敵を破壊する。
                // 破壊の具体的な処理は、DeleteEnemy関数で行うとする。
                if (type[i]==BREAKABLE) {
                    vit[i]-=str[j];
                    if (vit[i]<=0) DeleteEnemy(i);
                }
                // 敵に当たった武器は消す:
                // 具体的な処理はDeleteWeapon関数で行うとする。
                DeleteWeapon(j);
            }
        }
    }
}
```

敵の出現と消失

縦スクロールゲームの場合、基本的に敵は画面の上方から出現します（Fig. 6-4）。横スクロールゲームは画面の右方からです（Fig. 6-5）。

敵の動きはさまざまですが、もっとも多いパターンは出現した側とは反対側の画面端に向かって移動し、画面から出たら消えるというものです。縦スクロールゲームの場合には画面上方に出現して画面下方に向かい、横スクロールゲームの場合には画面右方に出現して画面左方に向かいます。なお、逆方向にスクロールするゲームでは出現と消失の位置が逆になります。

Fig. 6-4 敵の出現と消失（縦スクロールゲーム）

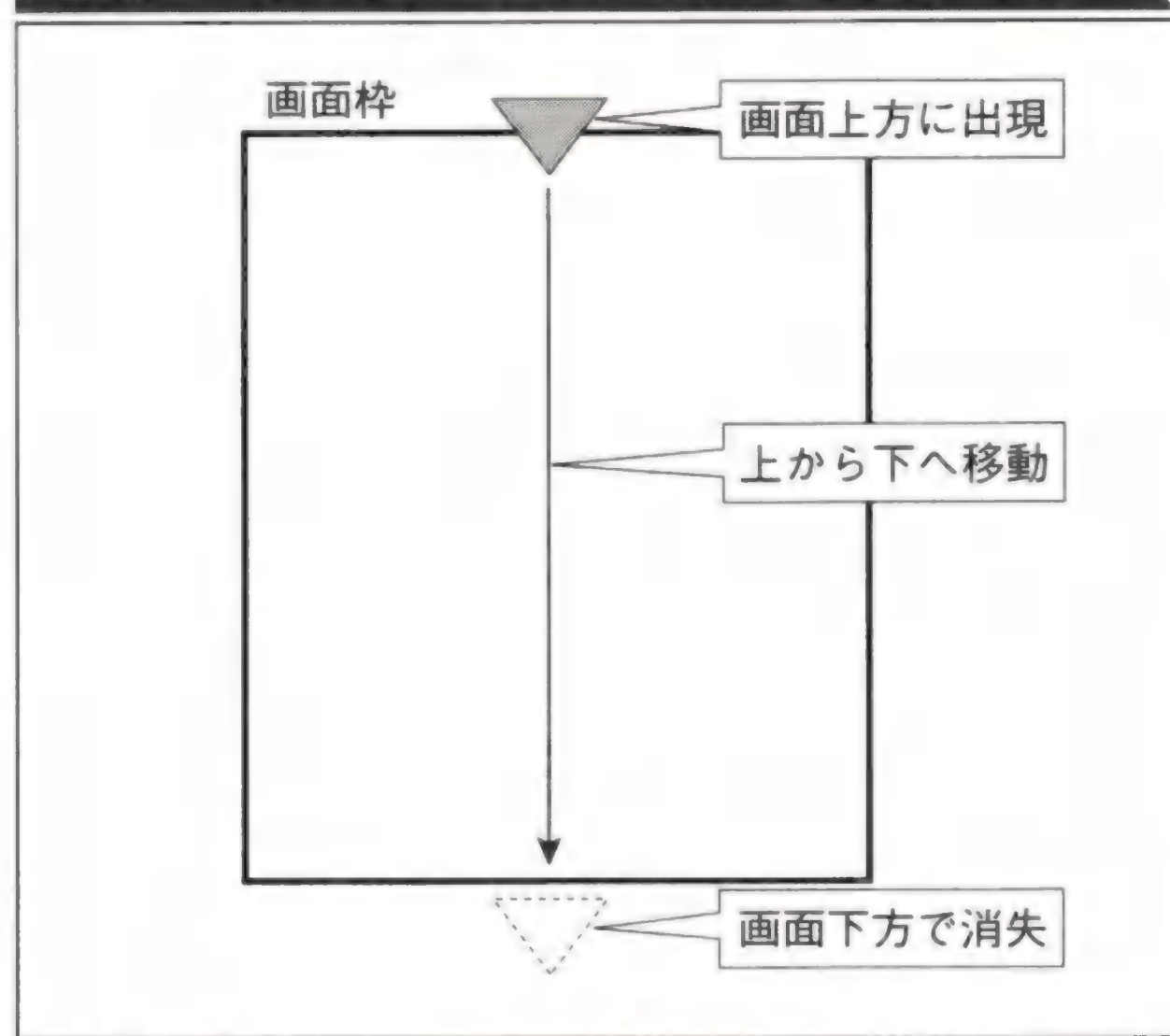



Fig. 6-5 敵の出現と消失（横スクロールゲーム）

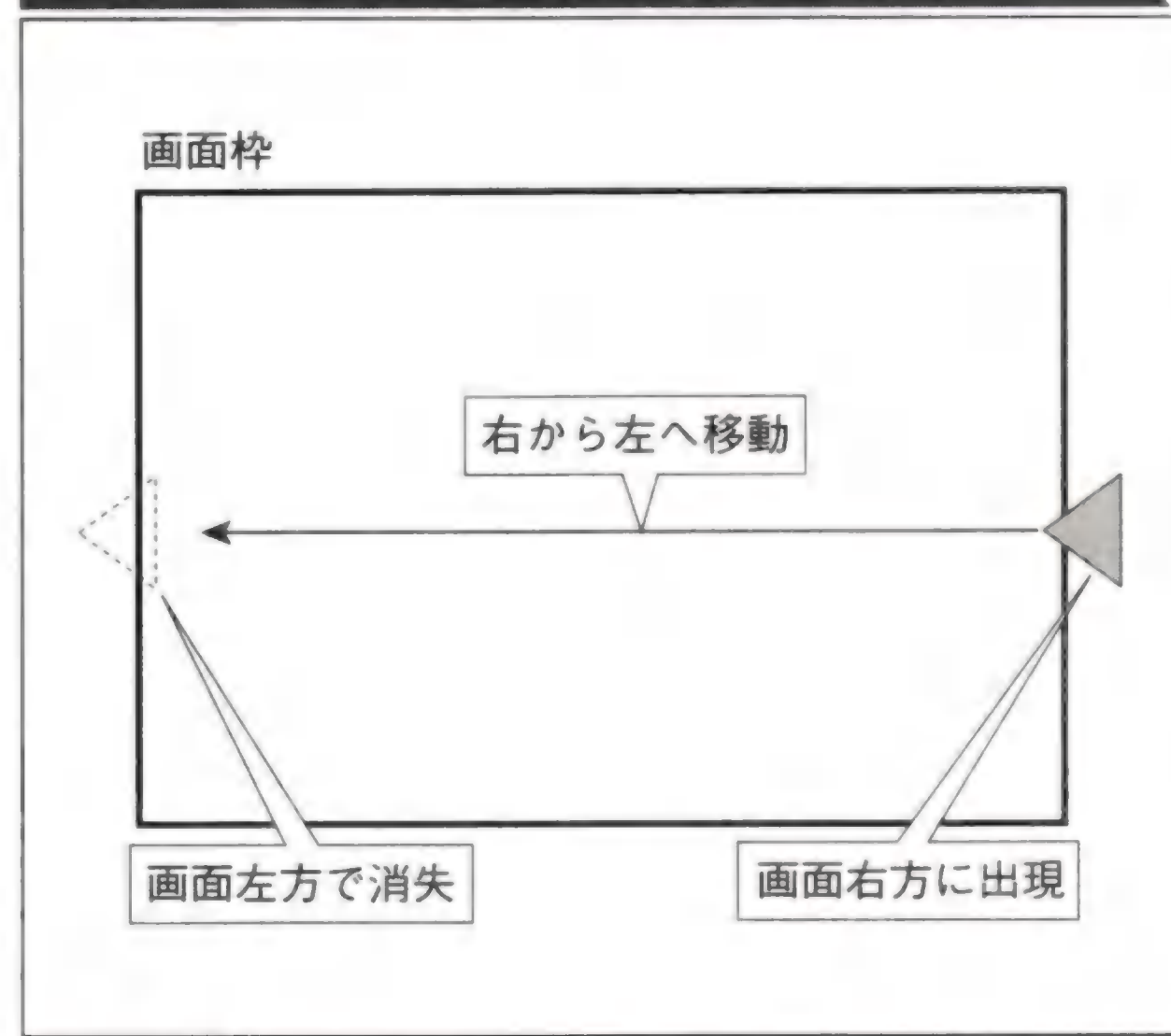
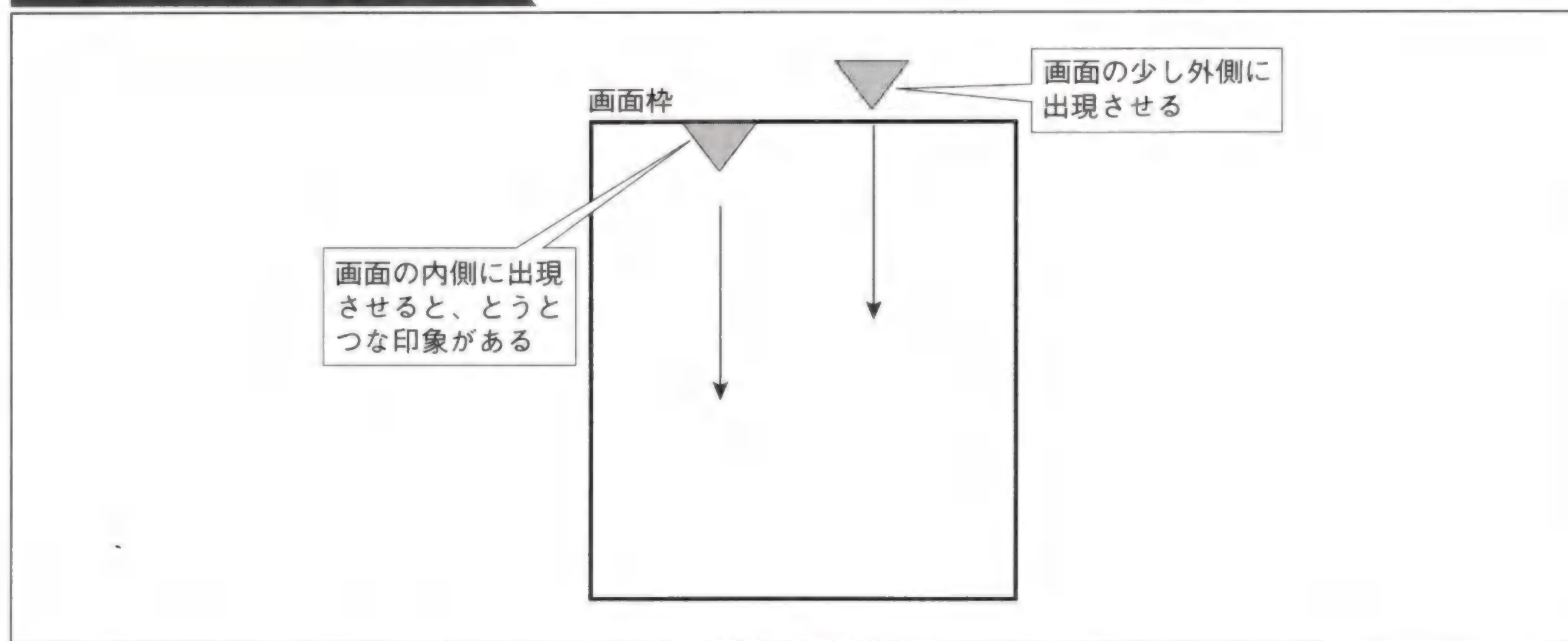



Fig. 6-6 敵を出現させる位置

敵を出現させるときには、画面の少し外側に出現させます（Fig. 6-6）。画面の内側に出現させると、何もない場所に敵が突然表示されて奇妙な感じがするからです。

敵を消去する際に画面の外側に出たかどうかを判定する方法は、「弾の消去」（→ P. 29）と同じです。たとえば敵の当たり判定の左上座標を（ex0, ey0）、右下座標を（ex1, ey1）、移動可能枠（画面枠）の左上座標を（sx0, sy0）、右下座標を（sx1, sy1）とすると、敵が画面の外側に出る条件は次のようになります。

(ex1<=sx0 || sx1<=ex0 || ey1<=sy0 || sy1<=ey0)

移動可能枠は実際の画面枠よりも少し広めにとっておきます。詳しい理由は「弾の消去」のところで解説しましたが、これは出現時に敵を消してしまわないための処置です。敵の出現と消失に関する処理はList 6-3にまとめました。

List 6-3 敵の出現と消失

```
void Enemy(
    int num_enemy,                  // 敵の数
    float ex0[], float ey0[],       // 敵の当たり判定の左上座標
    float ex1[], float ey1[],       // 敵の当たり判定の右下座標
    float sx0, float sy0,           // 移動可能枠（画面枠）の左上座標
    float sx1, float sy1            // 移動可能枠（画面枠）の右下座標
) {
    for (int i=0; i<num_enemy; i++) {

        // 敵の移動：
        // 移動の具体的な処理はDeleteEnemy関数で行うとする。
        MoveEnemy(i);

        // 敵の消失：
        // 敵が画面枠から出たかどうかを判定し、
        // 出ていたら消す。
        // 消失の具体的な処理はDeleteEnemy関数で行うとする。
        if (ex1[i]<=sx0 || sx1<=ex0[i] ||
            ey1[i]<=sy0 || sy1<=ey0[i]) {
            DeleteEnemy(i);
        }
    }
}
```

空中に現れる敵

画面の外側から近づいてくるのではなく、目の前の空中に突然現れる敵もいます（Fig. 6-7）。「ゼビウス（→ P. 329）」の「ザカート」などが有名な例です。

「ザカート」は本当に突然現れますが、今のハードウェアにはアルファ合成（アルファブレンディング）の機能があるので、これを使えばなめらかに敵を出現させることもできます（Fig. 6-8）。最初は敵を薄く表示しておき、だんだんと表示を濃くします。

空中に現れるのと逆の処理を行えば（Fig. 6-9）、空中で消える敵を作ることもできます。この場合もアルファ合成を使って、敵をだんだんと薄くしてから消せば、敵をなめらかに消失させることができます。

空中に現れる敵の出現と消失に関する処理をまとめたのがList 6-4です。出現中、活動中、消失中という3つの状態を用意して、処理を分岐することがポイントになります。

Fig. 6-7 空中に現れる敵

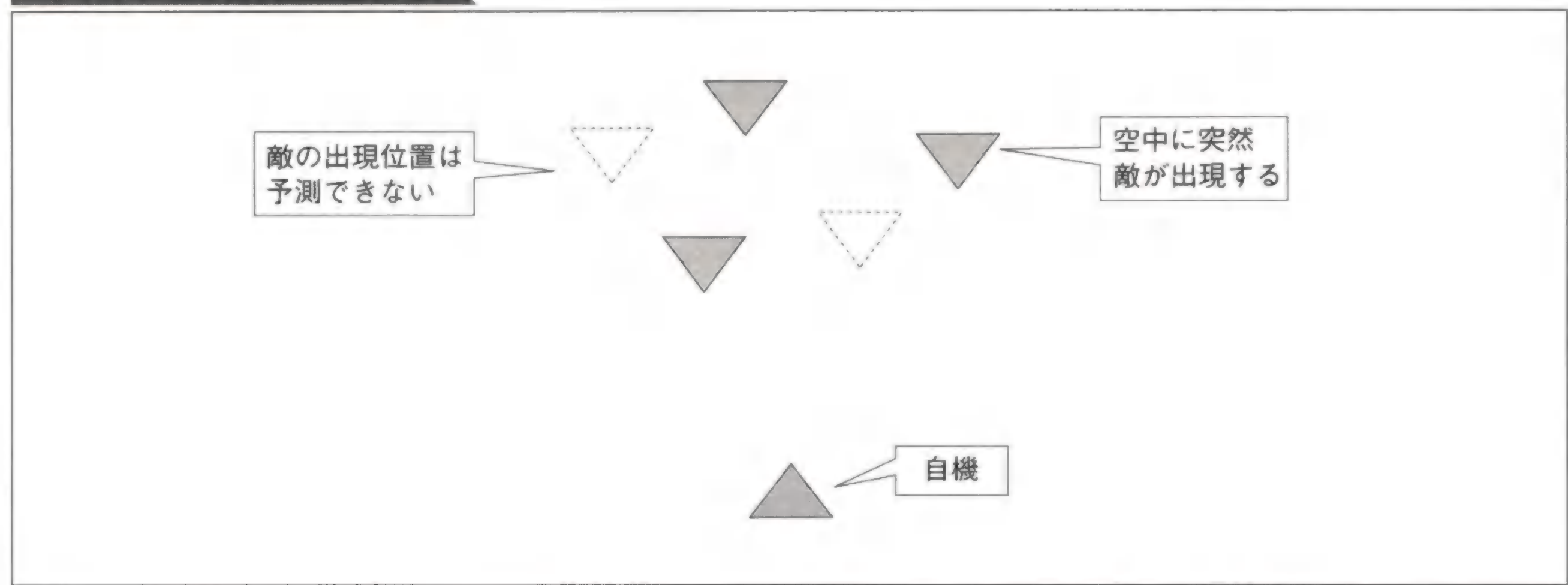



Fig. 6-8 アルファ合成を使った敵の出現

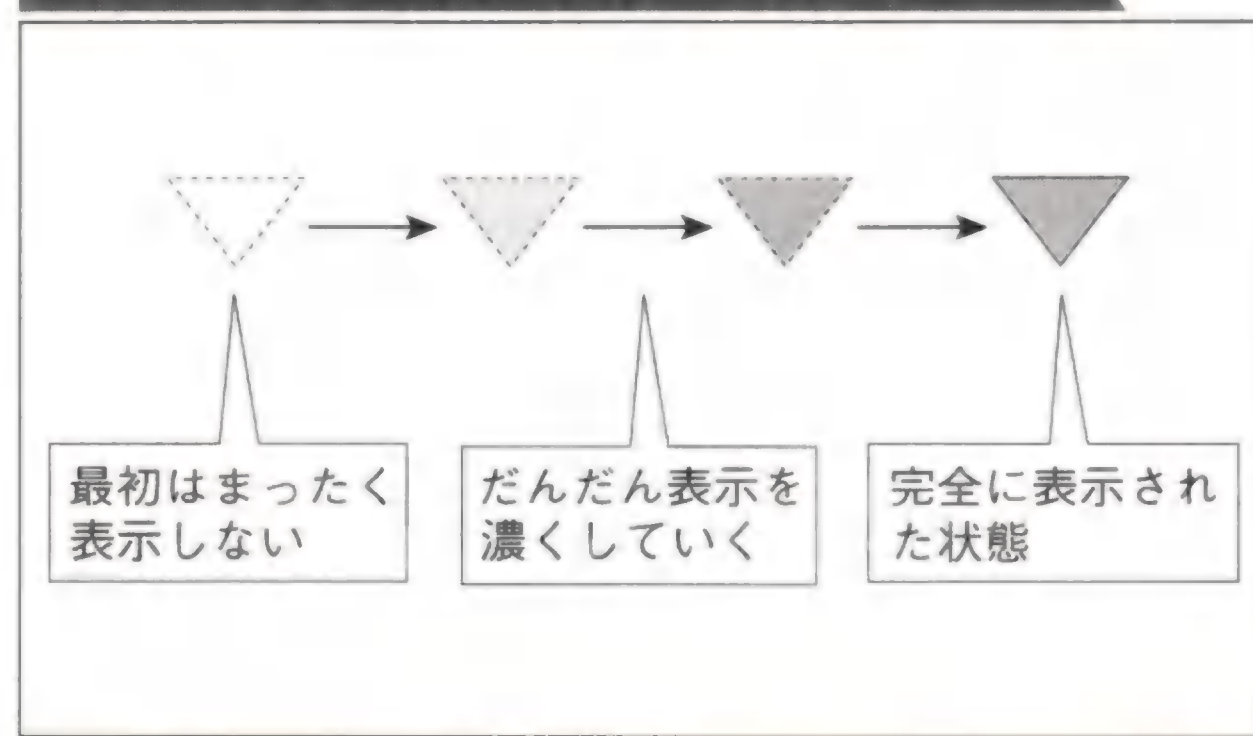
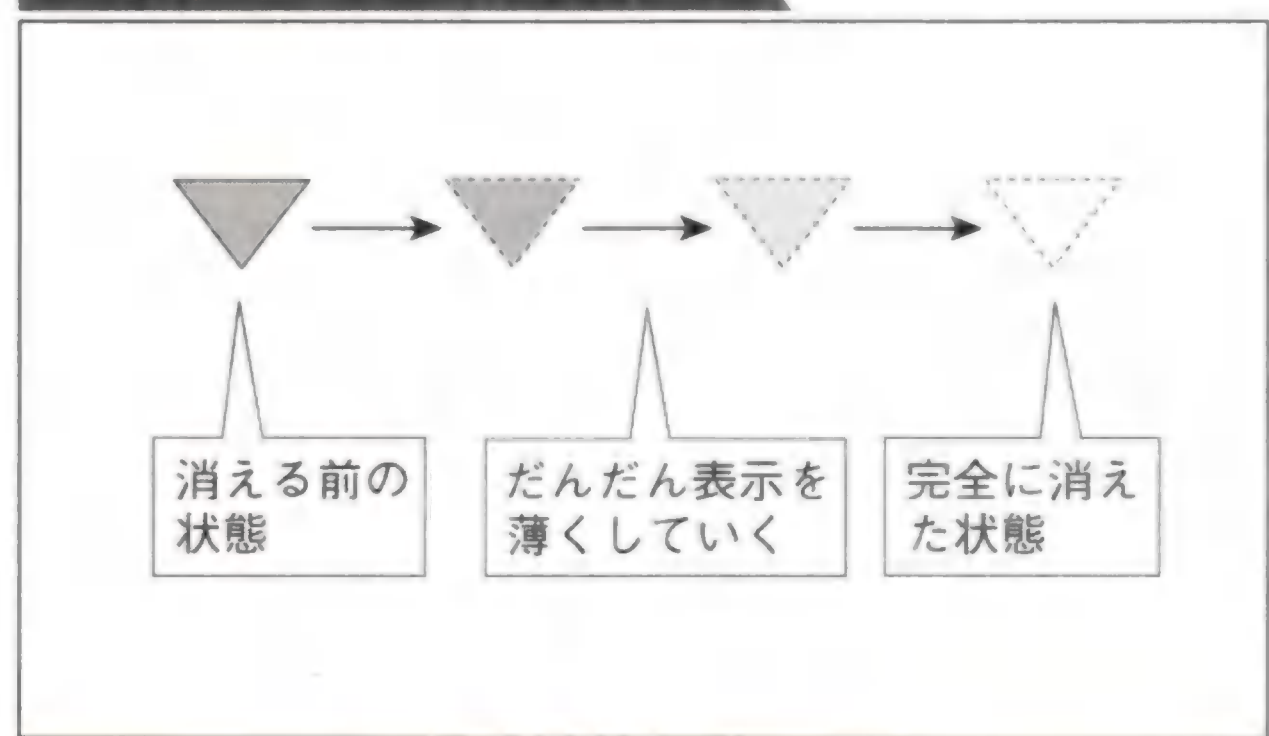



Fig. 6-9 空中で消える敵

サンプル

空中に現れる敵 → P. 320

List 6-4 空中に現れる敵

```
// 敵の状態を表す定数：
// 準備(出現前)、出現、活動、消失
typedef enum {
    READY, APPEAR, ACT, DISAPPEAR
} STATE_TYPE;

// 出現と消失にかかる時間、活動時間
#define TIME 30
#define ACT_TIME 180

// 空中に現れる敵の処理
void EmergingEnemy(
    int num_enemy,            // 敵の数
    STATE_TYPE state[],       // 敵の状態
    float ex[], float ey[],   // 敵の座標
    float alpha[],            // アルファ値
    int timer[]               // タイマー
) {
    // すべての敵に関するループ
    for (int i=0; i<num_enemy; i++) {

        // 状態に応じて分岐
        switch (state[i]) {

            // 準備(出現前)：
            // アルファ値とタイマーを初期化して、
            // 出現状態に移行する。
            case READY:
                state[i]=APPEAR;
                alpha[i]=0;
                timer[i]=TIME;
                break;

            // 出現：
            // アルファ値をだんだん大きくする。
            // タイマーが0になったら活動状態に移行する。
            case APPEAR:
                timer[i]--;
                alpha[i]=(float)(TIME-timer[i])/TIME;
                if (timer[i]==0) {
                    state[i]=ACT;
```

```
                timer[i]=ACT_TIME;
            }
            break;

        // 活動：
        // 移動や攻撃を行う。
        // タイマーが0になったら消失状態に移行する。
        // 移動や攻撃の具体的な処理は、
        // MoveEnemy関数で行うとする。
        case ACT:
            timer[i]--;
            MoveEnemy(i);
            if (timer[i]==0) {
                state[i]=DISAPPEAR;
                timer[i]=TIME;
            }
            break;

        // 消失：
        // アルファ値をだんだん小さくする。
        // タイマーが0になったら敵を消す。
        // 敵を消す具体的な処理は、
        // DeleteEnemy関数で行うとする。
        case DISAPPEAR:
            timer[i]--;
            alpha[i]=(float)timer[i]/TIME;
            if (timer[i]==0) {
                DeleteEnemy(i);
            }
        break;
    }

    // 敵の表示：
    // アルファ値に基づいてアルファ合成表示を行う。
    DrawEnemy(ex[i], ey[i], alpha[i]);
  }
}
```

敵の速さと硬さのバランス

敵の移動速度と耐久力とは、バランスを考えながら決める必要があります。たとえば、速く移動する敵を硬くしてしまうと、敵が画面外に出るまでに十分なダメージを与えることができないので、実質「破壊できない敵」になってしまいます。逆に、移動が遅い敵はある程度硬くしても大丈夫です（Fig. 6-10）。

空中に現れる敵についても同様です。出現してから消失するまでの時間が短い敵を硬くしてしまうと、敵が画面に滞在している間に破壊することができません。こういった場合には、敵の滞在時間を長くするか、もしくは耐久力を低くする必要があります。

敵の種類にかかわらず、要は「敵の耐久力」と「敵が画面に滞在する間に与えられるダメージ」のバランスがとれていればよいのです（Fig. 6-11）。たとえば、敵が画面に300フレーム（60フレーム毎秒のゲームで5秒間）滞在し、威力2のショットを毎フレーム撃てるとすれば、破壊できる敵の耐久力の最大値はだいたい600（＝300×2）になります。実際には最大のダメージを与えることは難しいので、最大値の1/2なり1/3なりの耐久力にしておきます。

Fig. 6-10 敵の速さと硬さのバランス

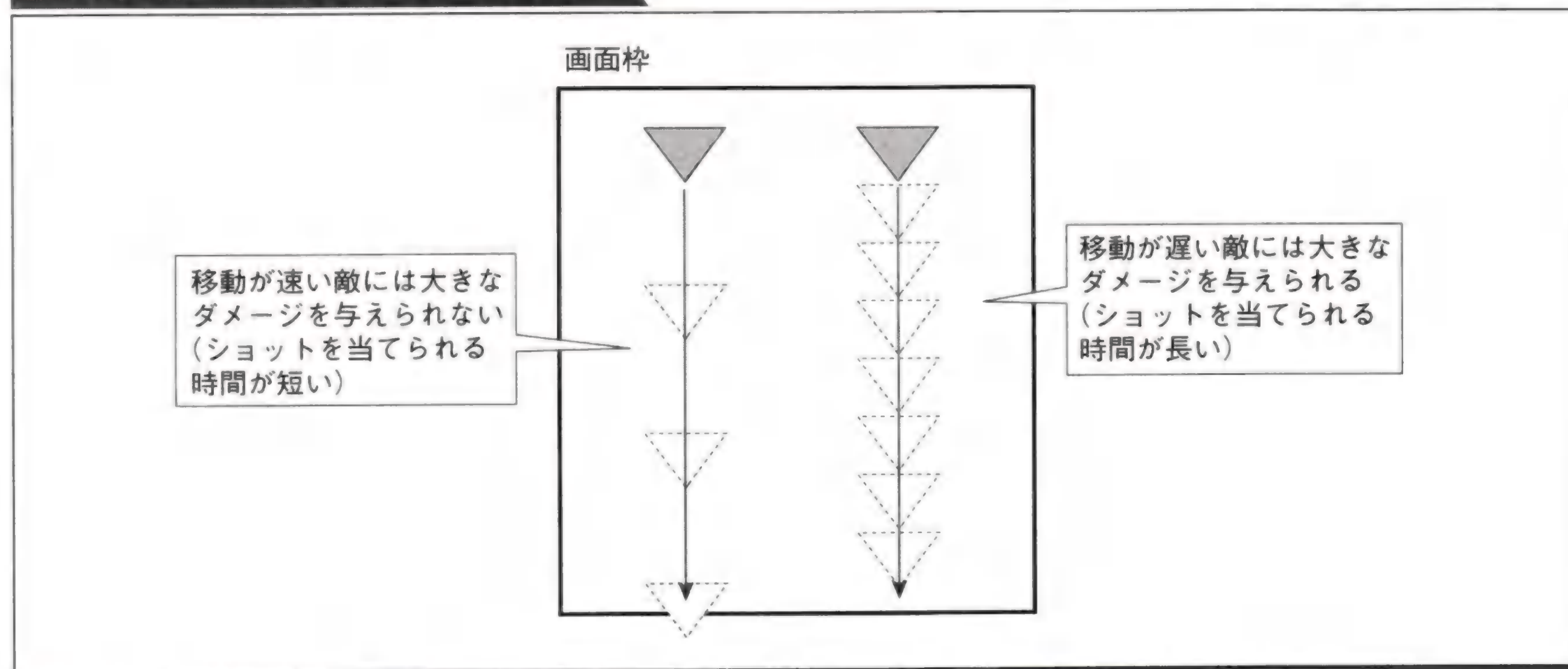
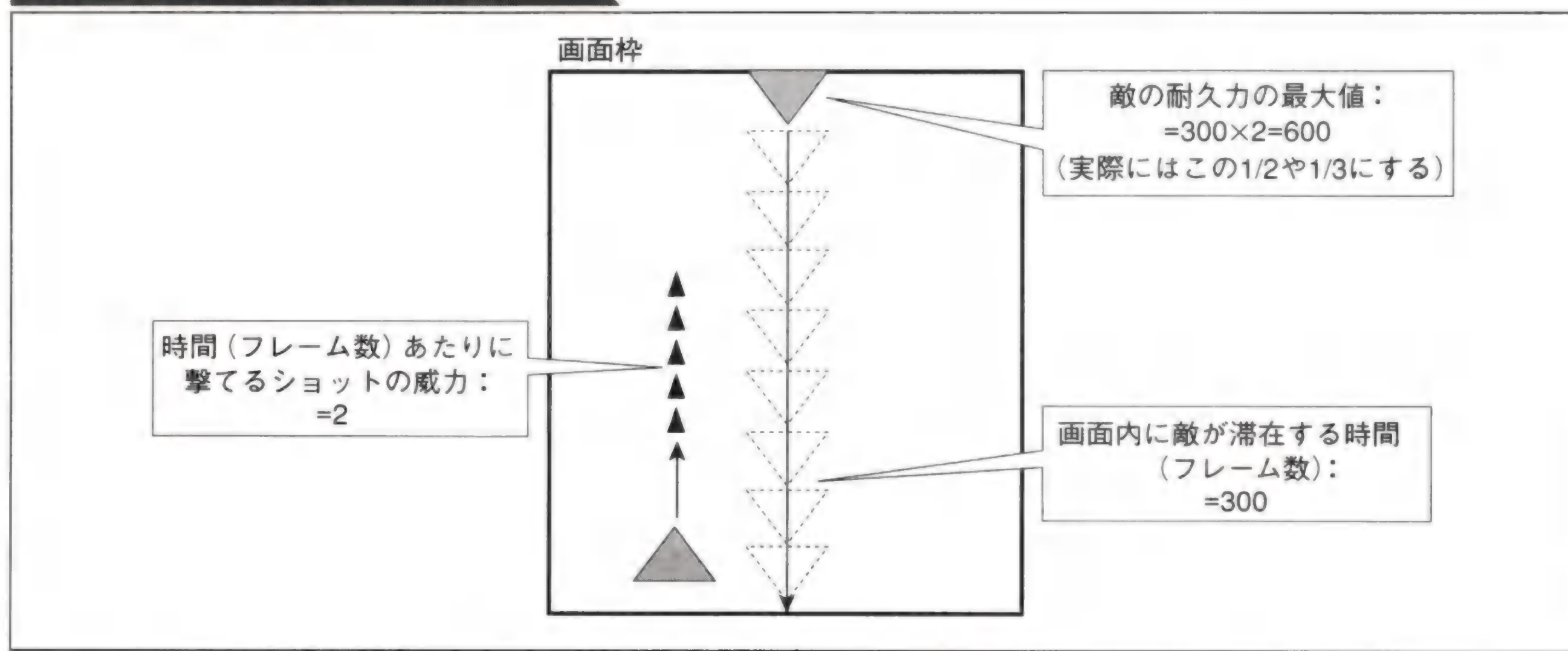
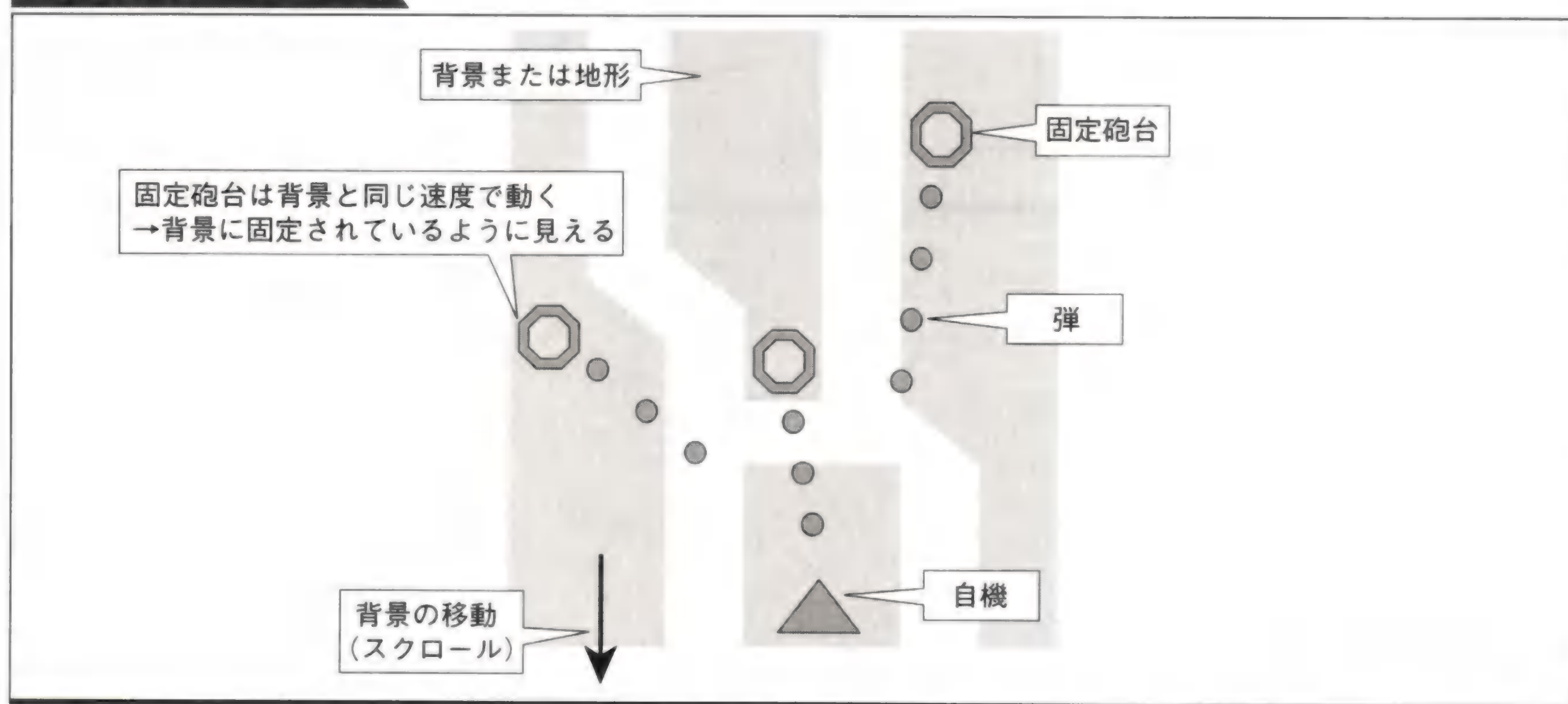



Fig. 6-11 敵の耐久力を決める方法

固定砲台

背景や地形に固定された砲台です。これは「背景と同じ速度で動く敵」だともいえます（Fig. 6-12）。背景の移動（スクロール）に合わせて砲台を動かせば、その砲台は背景に固定されているかのように見えます。

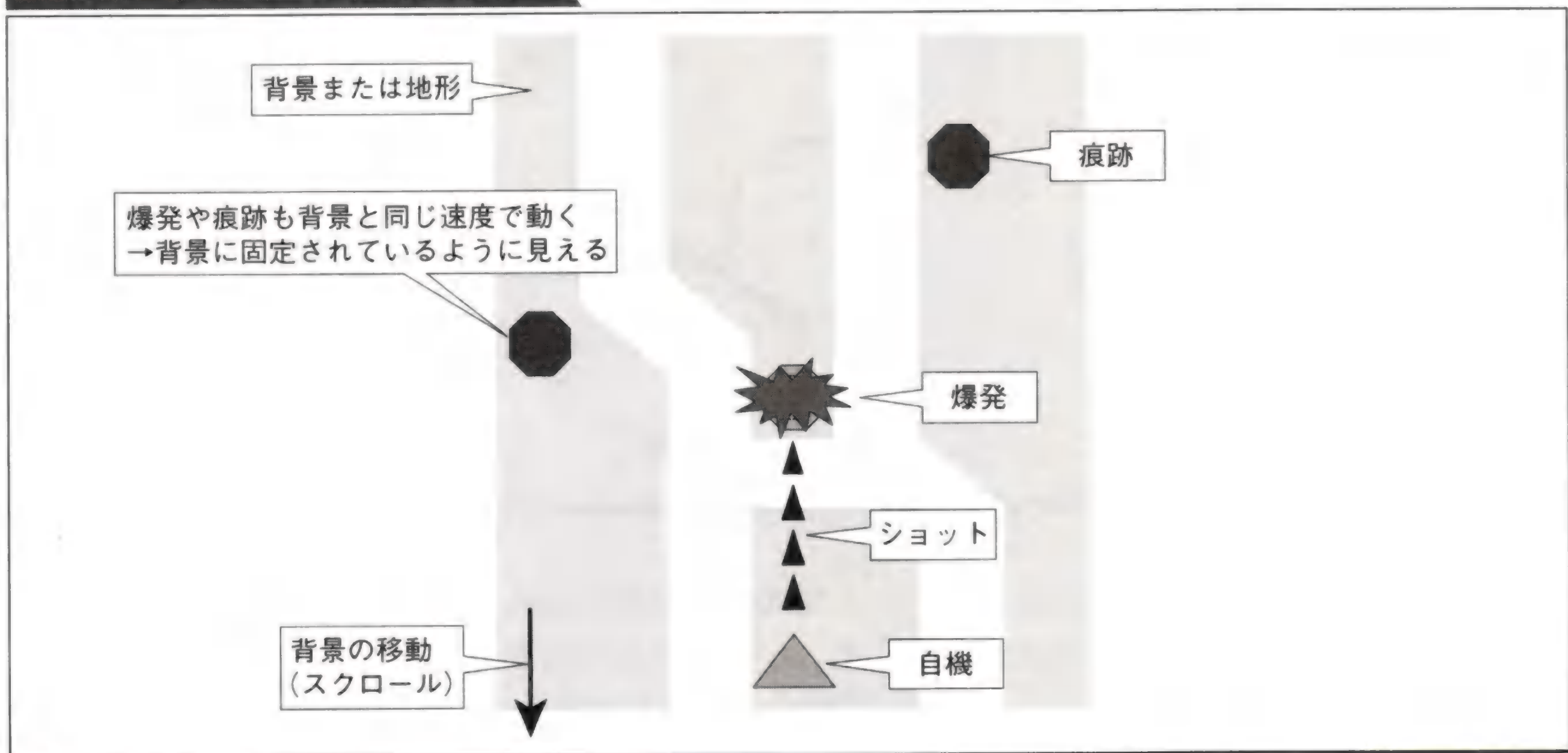
固定砲台が破壊されると、まず爆発が表示され、次に爆発の痕跡が残ります（Fig. 6-13）。この爆発と痕跡についても、地形と同じ速度で動かすことによって、地形に固定されているよう

Fig. 6-12 固定砲台

に見せる必要があります。

固定砲台に関する処理はList 6-5にまとめました。すべての砲台を背景と同じ速度で動かします。

サンプル

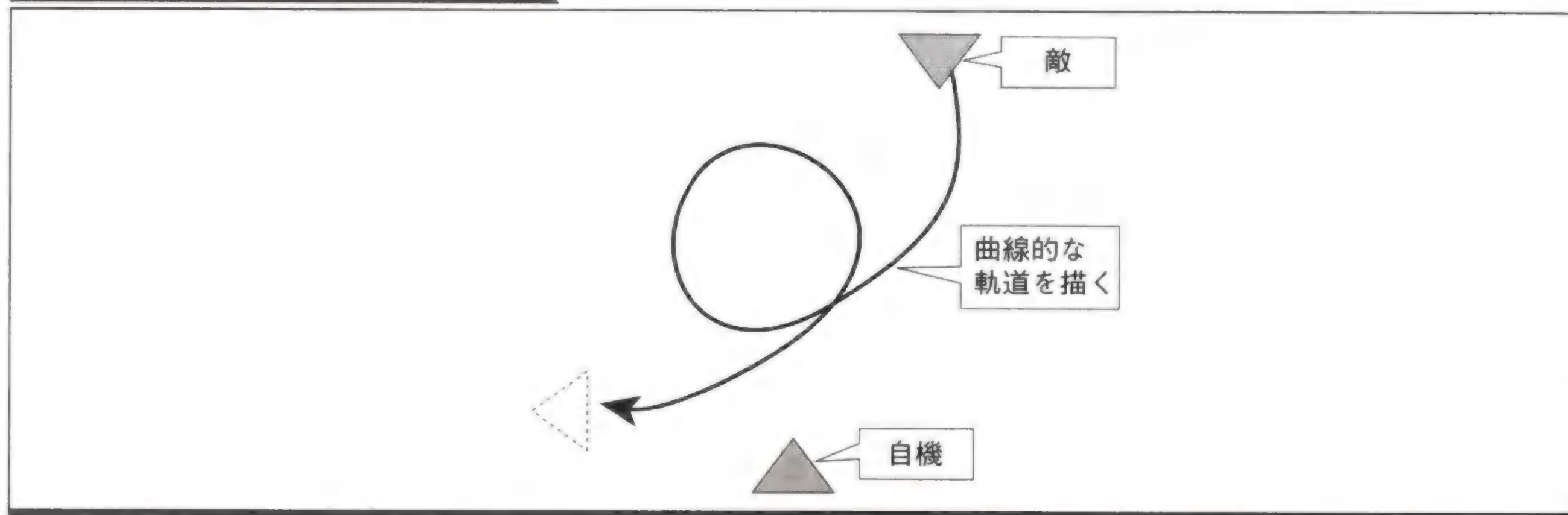
● 固定砲台 → P. 320

Fig. 6-13 固定砲台の爆発と痕跡

List 6-5 固定砲台

```
void MoveFixedBattery(
    int num_enemy,              // 敵（固定砲台）の数
    float ex[], float ey[],     // 敵（固定砲台）の座標
    float svx, float svy        // 背景の移動速度（スクロール速度）
) {
    // すべての固定砲台を背景と同じ速度で動かす
    for (int i=0; i<num_enemy; i++) {
        ex[i]+=svx;
        ey[i]+=svy;
    }
}
```

軌道を描いて飛ぶ敵

曲線的な軌道を描いてなめらかに飛ぶ敵です（Fig. 6-14）。「ギャラガ（→ P. 325）」や「ギャプラス（→ P. 326）」などでは、美しくもトリッキーな軌道を描きながら飛ぶ敵を楽しむことができます。

Fig. 6-14　軌道を描いて飛ぶ敵

最近のゲームでも、特に固定画面のゲームでは敵の軌道に凝ったものが多いようです。固定画面のゲームはスクロール画面のゲームに比べるとどうしても地味になりがちなので、そのぶん敵の動きを派手にする必要があります。逆にいえば、スクロールする画面に気をとられることなく敵の動きに集中できるので、敵の凝った動きを特にアピールしたいゲームは、あえて固定画面にするという選択も考えられます。

軌道を描いて飛ぶ敵を作るには、次の2つの方法があります。

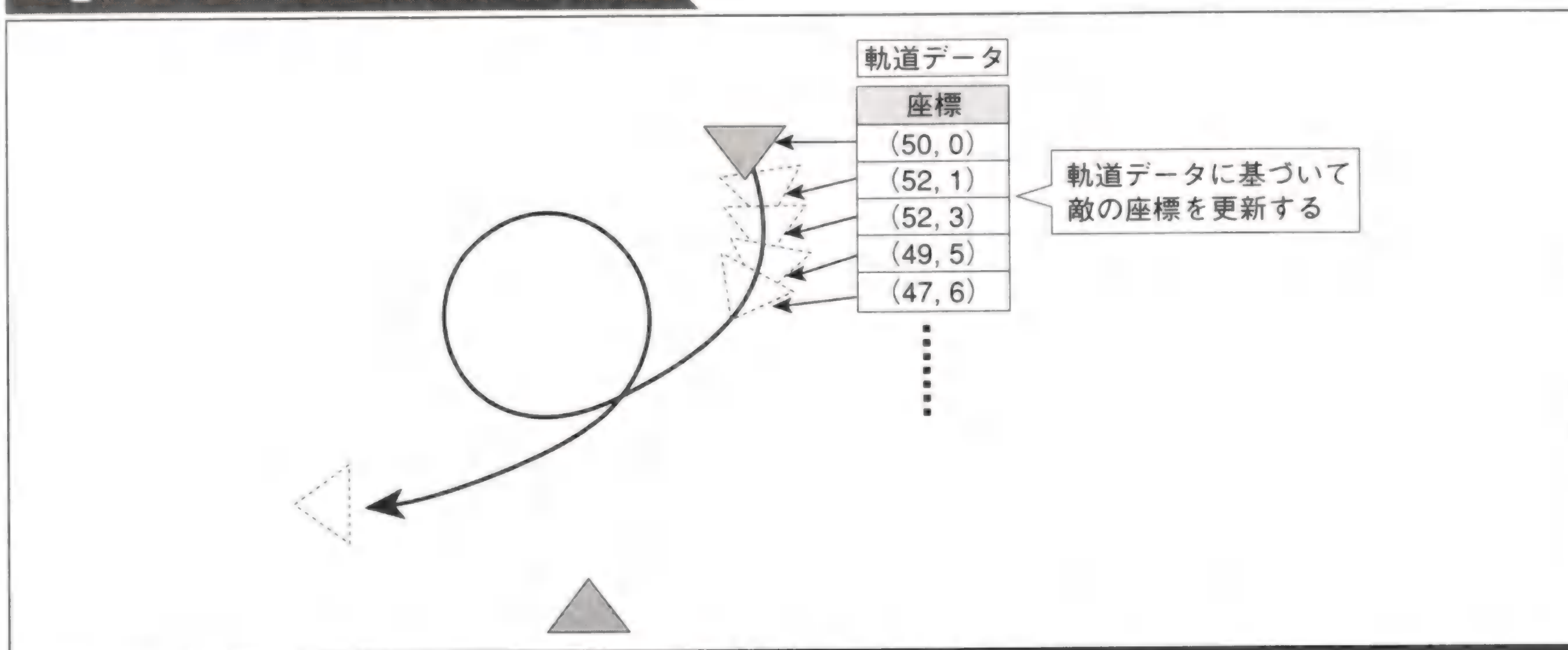
①軌道のデータを作っておく
②プログラムで軌道を生成する

①と②の方法を組み合わせて、「あらかじめ作ったデータを基本に、詳細な軌道はプログラムで生成する」というのも有効です。

■ 軌道データを作っておく

①の方法のように軌道のデータを作っておく場合、データを作るのはたいへんですが、作成したデータを使って敵を動かすのは簡単です（Fig. 6-15）。たとえば、軌道に沿った座標のデータを作っておきます。敵を動かすときには、データを順に読み出してきて、敵の座標を更新するだけです。この方法で敵を動かすプログラムはList 6-6のようになります。

①の方法でめんどうなのは、軌道データを用意する作業です。手作業で複雑な軌道を作るのは難しいので、軌道データ作成用のツールを自作するか、既存のツール（ドローツールなど）のデータを軌道データに変換するツールを作成しなければなりません。

Fig. 6-15 軌道データを使って敵を動かす

List 6-6 用意した軌道データを使って敵を動かす

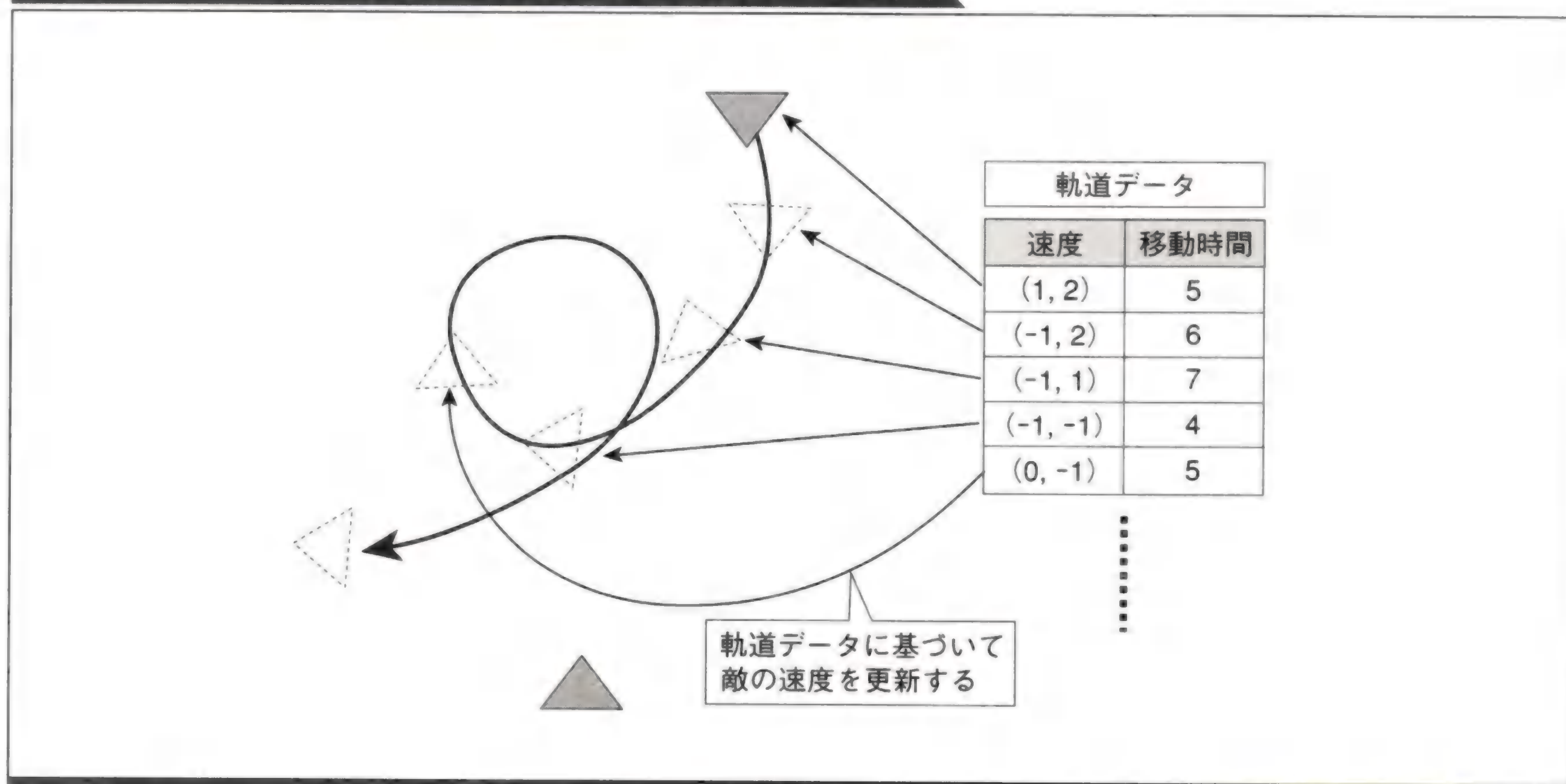
```
void OrbitalFlight(
    int num_enemy,              // 敵の数
    float ex[], float ey[],     // 敵の座標
    int index[],                // 軌道データを指すインデックス
    int num_orbit,              // 軌道データの要素数
    float ox[], float oy[]      // 軌道データ(座標)
) {
    for (int i=0; i<num_enemy; i++) {

        // 敵座標の更新:
        // 軌道データを読み出して、敵の新しい座標とする。
        ex[i]=ox[index[i]];
        ey[i]=oy[index[i]];

        // インデックス(データの読み出し位置)を進める:
        // 軌道データの最後に達したら敵を消す。
        // 敵を消す具体的な処理はDeleteEnemy関数で行うとする。
        index[i]++;
        if (index[i]==num_orbit) DeleteEnemy(i);
    }
}
```

■ 軌道データとプログラムの組み合わせ

①と②を組み合わせた方法では、軌道データを作るのは少し楽になります(Fig. 6-16)。たとえば、敵の速度と、その速度での移動時間とを組にしてデータにしておきます。プログラムはデータに基づいて敵の速度を更新し、決められた時間が過ぎたら次のデータを読み出します。これならば、軌道上の位置を1点ずつデータ化するよりも、だいぶデータが少なくてすみます。List 6-7はこの方法で敵を動かすプログラムです。

速度と移動時間ではなく、位置と移動時間や、加速度と移動時間などをデータにしておく方法もあります。加速度をデータ化すると、敵の加減速がなめらかになるので、曲線的な軌道が作りやすくなります。ただし、速度をデータ化する場合に比べて思いどおりの軌道を作るのは難しくなるので、軌道作成用のツールを作成しなければならないかもしれません。

Fig. 6-16 軌道データとプログラムを組み合わせて敵を動かす

速度	移動時間
(1, 2)	5
(-1, 2)	6
(-1, 1)	7
(-1, -1)	4
(0, -1)	5
⋮	

サンプル

軌道を描いて飛ぶ敵 → P. 320

List 6-7 軌道データとプログラムを組み合わせて敵を動かす

```
void OrbitalFlight2(
    int num_enemy,              // 敵の数
    float ex[], float ey[],     // 敵の座標
    int index[],                // 軌道データを指すインデックス
    int timer[],                // 移動時間を管理するタイマー
    int num_orbit,              // 軌道データの要素数
```

```
    float vx[], float vy[],  // 軌道データ(速度)
    int mov_time[]           // 軌道データ(移動時間)
) {
    for (int i=0; i<num_enemy; i++) {

        // 敵座標の更新:
        // 軌道データ(速度)を使って敵を動かす。
        ex[i]+=vx[index[i]];
        ey[i]+=vy[index[i]];

        // タイマーを進める:
        // タイマーが軌道データの移動時間を超えたら、
        // インデックスを進める。
        timer[i]++;
        if (timer[i]==mov_time[i]) {
            timer[i]=0;

            // インデックスを進める:
            // 軌道データの最後に達したら敵を消す。
            // 敵を消す具体的な処理はDeleteEnemy関数で行うとする。
            index[i]++;
            if (index[i]==num_orbit) DeleteEnemy(i);
        }
    }
}
```

敵の編隊

多数の敵が編隊を組んで飛行するものです(Fig. 6-17)。特に「軌道を描いて飛ぶ敵」(→ P. 226)を編隊にしたものは、視覚的に大きなインパクトがあります。編隊を組んだ敵が美しい軌道を描くゲームとしては「ギャラガ(→ P. 325)」や「ギャプラス(→ P. 326)」などが有名です。

敵の動きが単純なときには、簡単に編隊を実現する方法があります。同じ動きをする複数の敵を、同じ位置から、少しずつ時間をずらして発生させればよいのです。

一方、複雑な動きをする敵(たとえば誘導弾のような動きをする敵など)に編隊を組ませるときには、「誘導レーザー」(→ P. 46)に似た方法が使えます。前を飛ぶ敵の背後に、少し間隔を空けて別の敵を飛ばせばよいのです。同じ方法で敵を連ねていけば、いくらでも長い編隊を作ることができます。

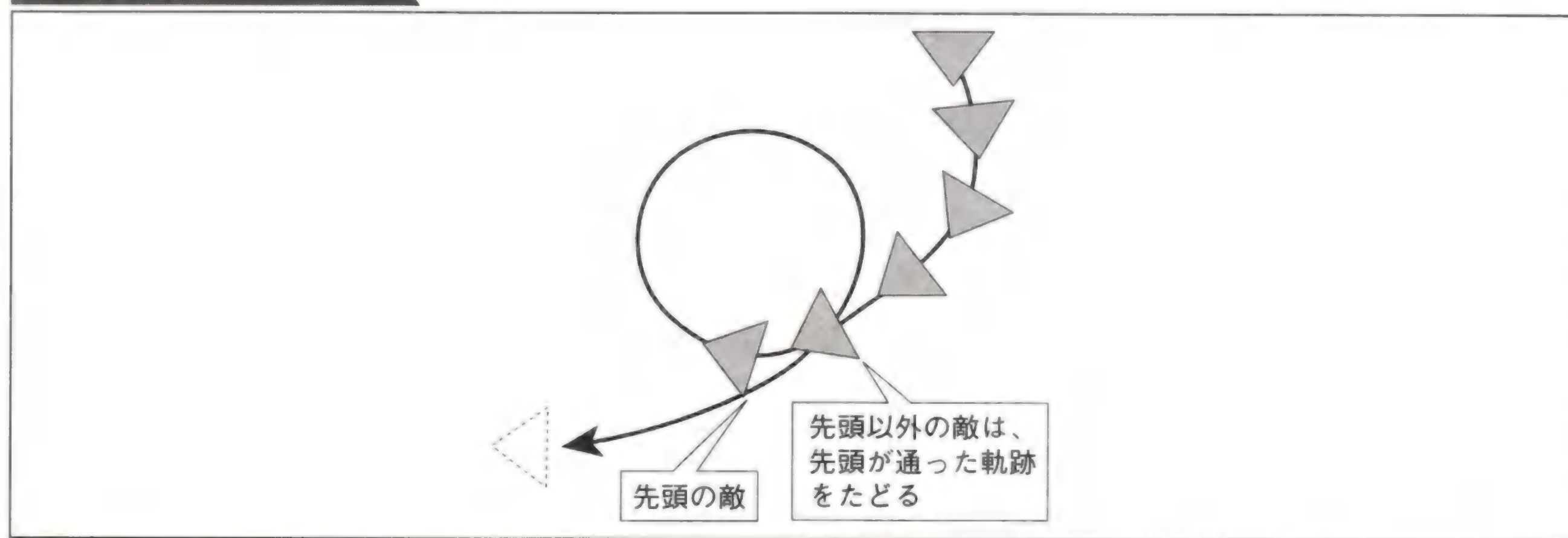
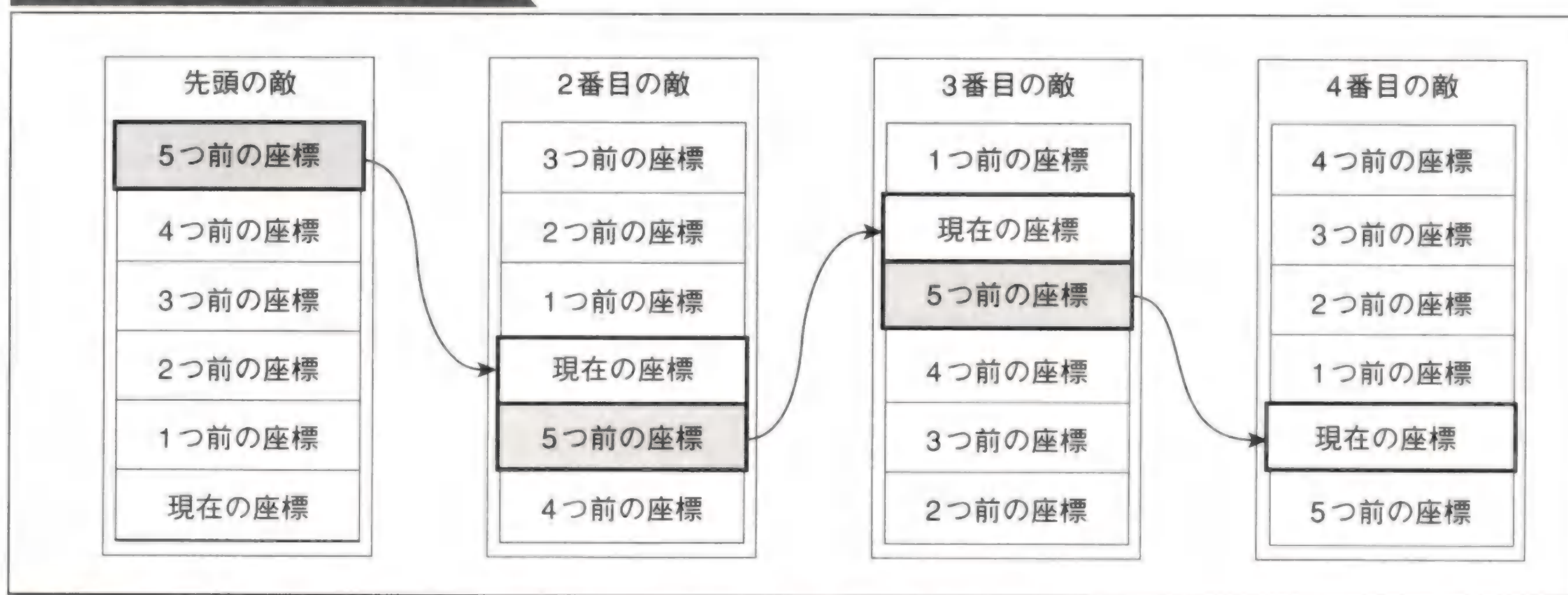
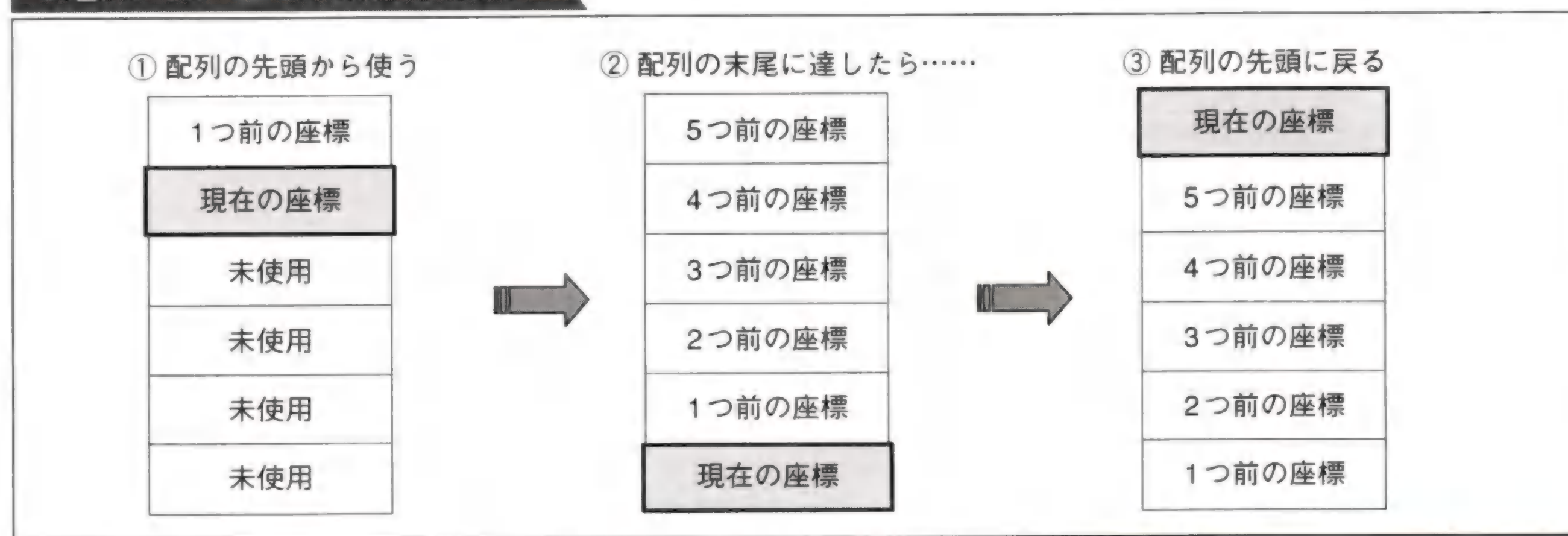
Fig. 6-17 敵の編隊

具体的には、それぞれの敵が通った座標(必要ならば角度も)を記録しておきます(Fig. 6-18)。このとき注意しなければならないのは、誘導レーザーの場合にはレーザーの各部分が密集していますが、編隊の場合には少し間隔が空いているということです。間隔を空けるためには、1つ前に通った座標だけではなく、たとえば5つや10まで前の座標を保存しておく必要があります。前の敵を追う後ろの敵は、保存されているなかでもっとも古い座標を使います。

Fig. 6-18では、それぞれの敵について5つ前までの座標を保存しています。先頭以外の敵は、自分の1つ前の敵が保存しているなかからもっとも古い座標(ここでは5つ前の座標)を読み込んで、自分の新しい座標にします。結果として、それぞれの敵が少しずつ間隔を空けながら先頭の敵を追うことになります。

Fig. 6-18 通った座標の記録

■ 座標の保存

古い座標を保存するには配列を使います。配列の長さにはかぎりがあるので、「オプション」(→ P. 104)と同じように、配列を循環させて使います(Fig. 6-19)。最初は配列の先頭から始めて、末尾に向かって使います(Fig. 6-19-①)。末尾に達したら(Fig. 6-19-②)、次は先頭に戻って使います(Fig. 6-19-③)。なお、このように循環させながら使うデータ保存領域のことを「リングバッファ」と呼ぶことがあります。

List 6-8は編隊の移動に関するプログラムです。ここでは敵の構造体のなかに、古い座標を保存するための配列を用意しました。

Fig. 6-19 配列を循環させて使う

List 6-8 編隊の移動

```
// 敵の間隔
#define INTERVAL 6

// 敵の情報(構造体)
typedef struct ENEMY_STRUCT {
    float X, Y;                  // 現在の座標
    float OldX[INTERVAL];        // 古いX座標
    float OldY[INTERVAL];        // 古いY座標
    int Index;                   // もっとも古い座標を指すインデックス
    struct ENEMY_STRUCT* Prec;   // 1つ前の敵
} ENEMY;

// 編隊飛行の処理
void Formation(
    int num_enemy,   // 敵の数
    ENEMY* enemy[]   // 敵の情報(構造体へのポインタ)
) {
    // 敵を動かす
    int i;
```

```
    for (i=0; i<num_enemy; i++) {
        ENEMY* self=enemy[i];
        ENEMY* prec=self->Prec;

        // 先頭以外の敵を動かす：
        // 1つ前の敵が保存している古い座標のうち、
        // もっとも古いものを自分の座標にする。
        if (prec) {
            self->X=prec->OldX[prec->Index];
            self->Y=prec->OldY[prec->Index];
        }

        // 先頭の敵を動かす：
        // 具体的な処理はMoveEnemy関数で行うとする。
        else {
            MoveEnemy(i);
        }
    }

    // 古い座標を記録する：
    // もっとも古い座標を新しい座標で上書きし、
    // インデックスを更新する。
    for (i=0; i<num_enemy; i++) {
        ENEMY* self=enemy[i];
        self->OldX[self->Index]=self->X;
        self->OldY[self->Index]=self->Y;
        self->Index=(self->Index+1)%INTERVAL;
    }
}
```

■ 編隊の出現

敵の編隊を出現させる方法は、誘導レーザーの発射方法に似ています。まず編隊の先端となる敵を出現させ、次にそのあとを追う敵を出現させます。

List 6-9は編隊を出現させるプログラムです。先端となる敵には、それより前の敵がいないので、参照先をNULLとします。

サンプル

敵の編隊　→ P. 320

List 6-9 編隊の出現

```
#include <stdlib.h>

// 敵の間隔
#define INTERVAL 6

// 敵の情報(構造体)
typedef struct ENEMY_STRUCT {
    float X, Y;                    // 現在の座標
    float OldX[INTERVAL];          // 古いX座標
    float OldY[INTERVAL];          // 古いY座標
    int Index;                     // もっとも古い座標を指すインデックス
    struct ENEMY_STRUCT* Prec;     // 1つ前の敵
} ENEMY;

// 編隊の生成
void CreateFormation(
    float x, float y,   // 生成地点の座標
    int count           // 編隊を構成する敵の数
) {
    ENEMY* enemy;       // 敵を表す構造体へのポインタ
    ENEMY* prec=NULL;   // 1つ前の敵を指すポインタ

    // 編隊を構成する敵を作る:
    // 敵の構造体を確保し、座標を初期化する。
    // 構造体確保の具体的な処理はNewEnemy関数で行うとする。
    for (int i=0; i<count; i++, prec=enemy) {
        enemy=NewEnemy();
        enemy->X=x;
        enemy->Y=y;
        for (int j=0; j<INTERVAL; j++) {
            enemy->OldX[j]=x;
            enemy->OldY[j]=y;
        }
        enemy->Index=0;

        // 先頭以外の敵は1つ前の敵を参照し、
        // 先端の敵はNULLを参照する。
        enemy->Prec=prec;
    }
}
```

複雑な形の当たり判定

敵の当たり判定は、弾や自機と同様に矩形（四角形）を使うのが基本です（Fig. 6-20）。敵の形と当たり判定が厳密に一致しているわけではありませんが、比較的小さなザコ敵や、形が矩形に近い敵についてはこれでも問題は生じません。

しかし、大きな敵や複雑な形をした敵の場合には、矩形の当たり判定では不自然になってしまうことがあります（Fig. 6-21）。特に敵の一部がへこんでいたり、穴が開いていたりする場合には、当たり判定を矩形にすることはできません。

Fig. 6-20 基本の当たり判定

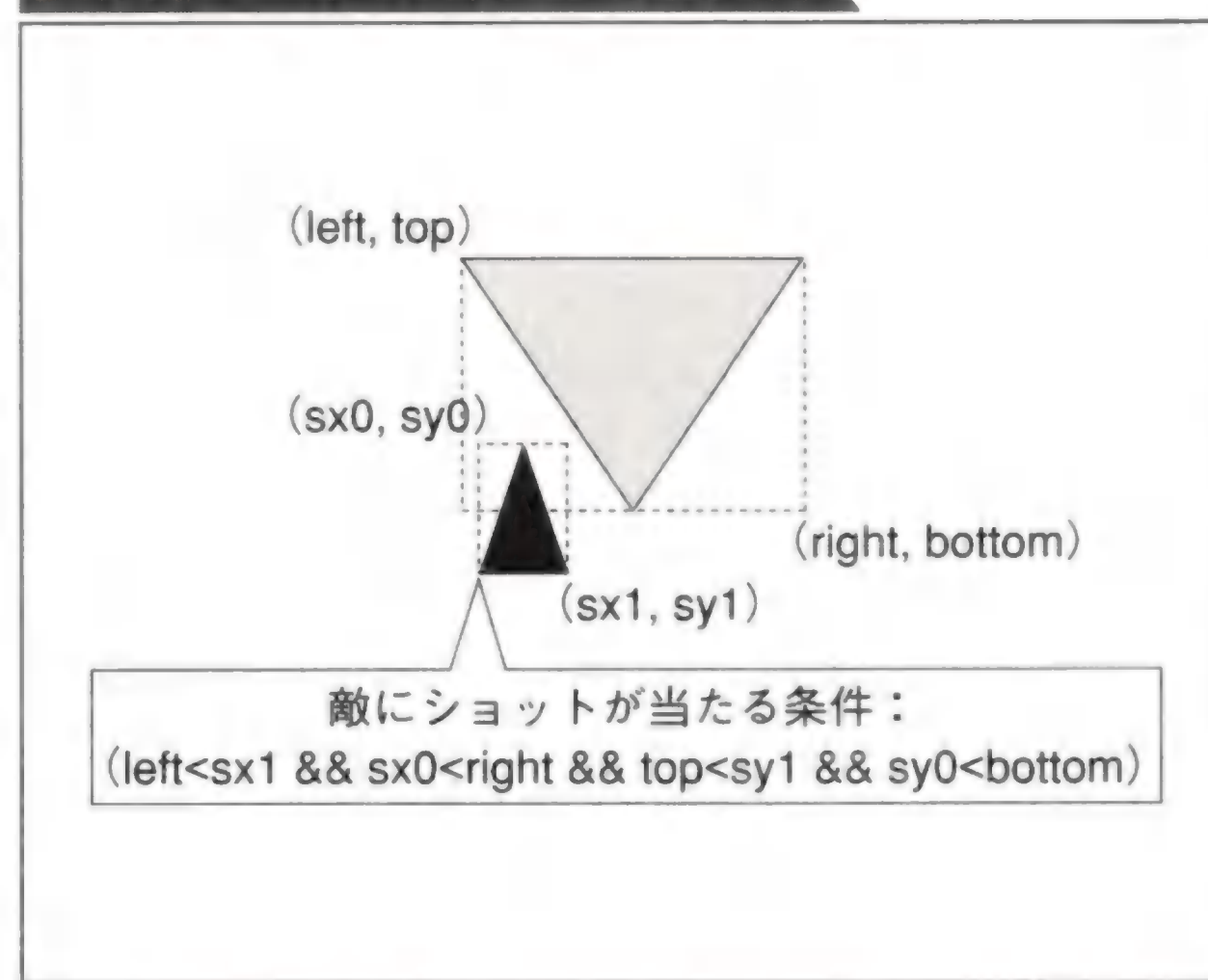
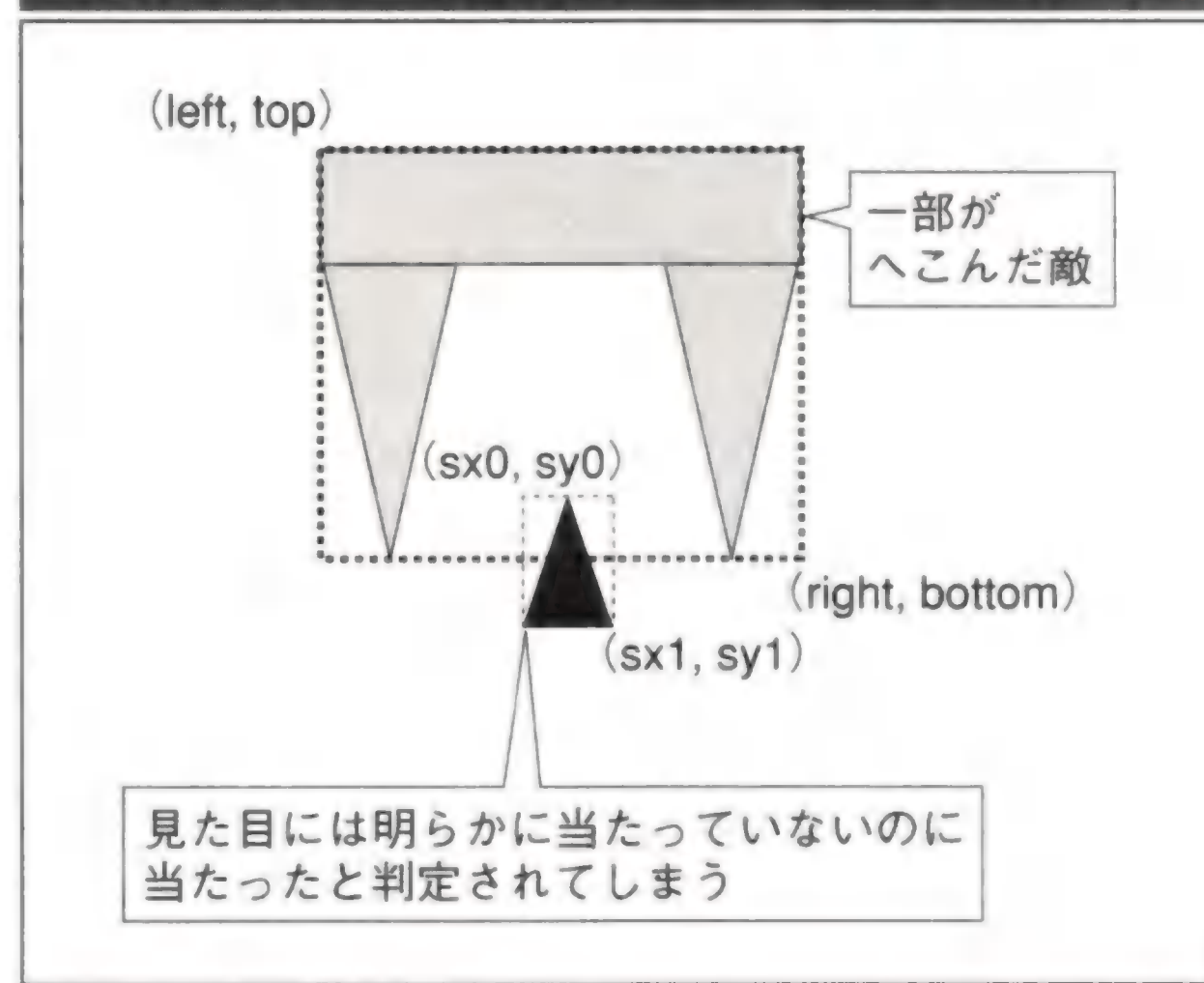



Fig. 6-21 矩形の当たり判定では問題がある場合

こういった場合には、複数の矩形を組み合わせて当たり判定を作るのが一般的です（Fig. 6-22）。そして、複数の矩形に関して順に当たり判定処理を行います。いずれかの矩形について条件が成立したら、その敵に当たったということになります。

複雑な形をした物体同士が衝突したかどうかを判定するには、双方のすべての矩形の組み合わせについて当たり判定処理を行います（Fig. 6-23）。たとえば、物体Aと物体Bがあり、それぞれが3つずつの当たり判定から構成されていたら、9通り（=3×3）の組み合わせに関する処理が必要です。

このように複雑な物体同士の当たり判定処理では、処理しなければならない条件式の数が急に増えます。実際のゲームではできるだけ物体の当たり判定を簡単にして、「矩形」対「矩形」の当たり判定か、せいぜい「矩形」対「複雑な形」の当たり判定にとどめるのがよい方法です。

List 6-10は複雑な形の当たり判定についてまとめたプログラムです。「矩形」対「矩形」、「矩形」対「複雑な形」、「複雑な形」対「複雑な形」という3種類の処理を用意しました。

Fig. 6-22 複数の矩形を組み合わせた当たり判定

矩形の当たり判定#0：
(left [0], top [0]) ～ (right [0], bottom [0])

矩形の当たり判定#1：
(left [1], top [1]) ～
(right [1], bottom [1])

矩形の当たり判定#2：
(left [2], top [2]) ～
(right [2], bottom [2])

(sx0, sy0)

(sx1, sy1)

敵にショットが当たる条件：
(left [0] <sx1 && sx0<right [0] && top [0] <sy1 && sy0<bottom [0]) ||
(left [1] <sx1 && sx0<right [1] && top [1] <sy1 && sy0<bottom [1]) ||
(left [2] <sx1 && sx0vright [2] && top [2] <sy1 && sy0<bottom [2])

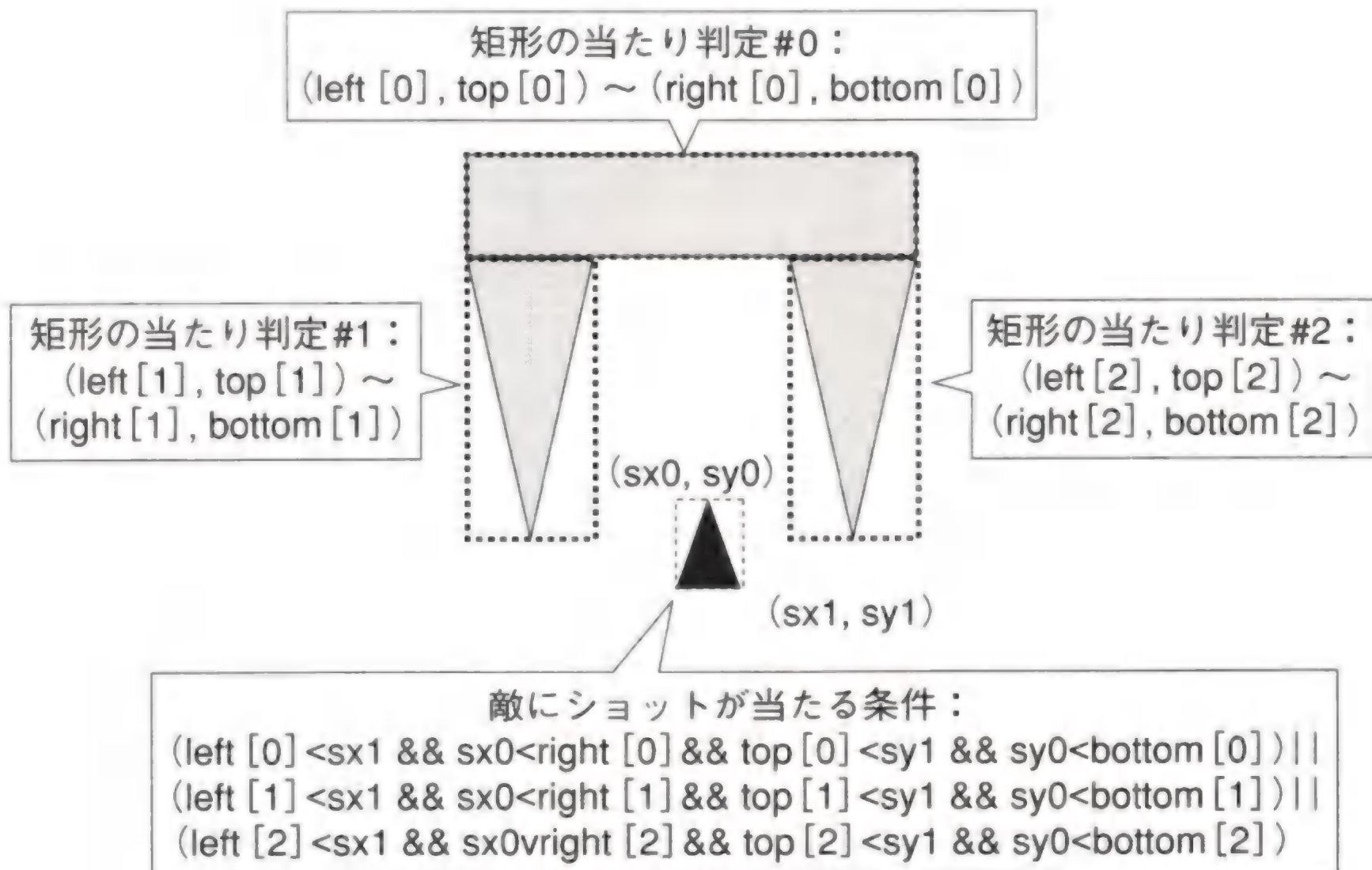



Fig. 6-23 複雑な形をした物体同士の当たり判定処理

物体aの当たり判定#0：
(left [0], top [0]) ～
(right [0], bottom [0])

物体aの当たり判定#1：
(left [1], top [1]) ～
(right [1], bottom [1])

物体aの当たり判定#2：
(left [2], top [2]) ～
(right [2], bottom [2])

物体bの当たり判定#1：
(left [1], top [1]) ～
(right [1], bottom [1])

物体bの当たり判定#2：
(left [2], top [2]) ～
(right [2], bottom [2])

物体bの当たり判定#0：
(left [0], top [0]) ～
(right [0], bottom [0])

物体aと物体bが衝突する条件：
(a->left [0] <b->right [0] && b->left [0] <a->right [0] &&
a->top [0] <b->bottom [0] && b->top [0] <a->bottom [0]) ||
(a->left [0] <b->right [1] && b->left [1] <a->right [0] &&
a->top [0] <b->bottom [1] && b->top [1] <a->bottom [0]) ||
⋮
(a->left [2] <b->right [2] && b->left [2] <a->right [2] &&
a->top [2] <b->bottom [2] && b->top [2] <a->bottom [2])

List 6-10 複雑な形の当たり判定

```
// 矩形の当たり判定を表す構造体
typedef struct {
    float Left, Top, Right, Bottom;
} HIT;

// 複雑な当たり判定を表す構造体
#define MAX_HIT 32
typedef struct {
    HIT* Hit[MAX_HIT];  // 矩形の当たり判定
    int Num;            // 当たり判定の個数
} COMPLEX_HIT;

// 「矩形」対「矩形」の基本的な当たり判定処理
bool IsHit(HIT* a, HIT* b) {
    return
        a->Left<b->Right && b->Left<a->Right &&
        a->Top<b->Bottom && b->Top<a->Bottom;
}

// 「矩形」対「複雑な形」の当たり判定処理:
// 複数の矩形に対して順に当たり判定処理を行い、
// いずれかについて条件が成立したら「当たり」とする。
bool IsHit(COMPLEX_HIT* a, HIT* b) {
    for (int i=0; i<a->Num; i++) {
        if (IsHit(a->Hit[i], b)) return true;
    }
    return false;
}

// 「複雑な形」対「複雑な形」の当たり判定処理:
// すべての組み合わせについて当たり判定処理を行い、
// いずれかの組み合わせで条件が成立したら「当たり」とする。
bool IsHit(COMPLEX_HIT* a, COMPLEX_HIT* b) {
    for (int i=0; i<a->Num; i++) {
        for (int j=0; j<b->Num; j++) {
            if (IsHit(a->Hit[i], b->Hit[j])) return true;
        }
    }
    return false;
}
```

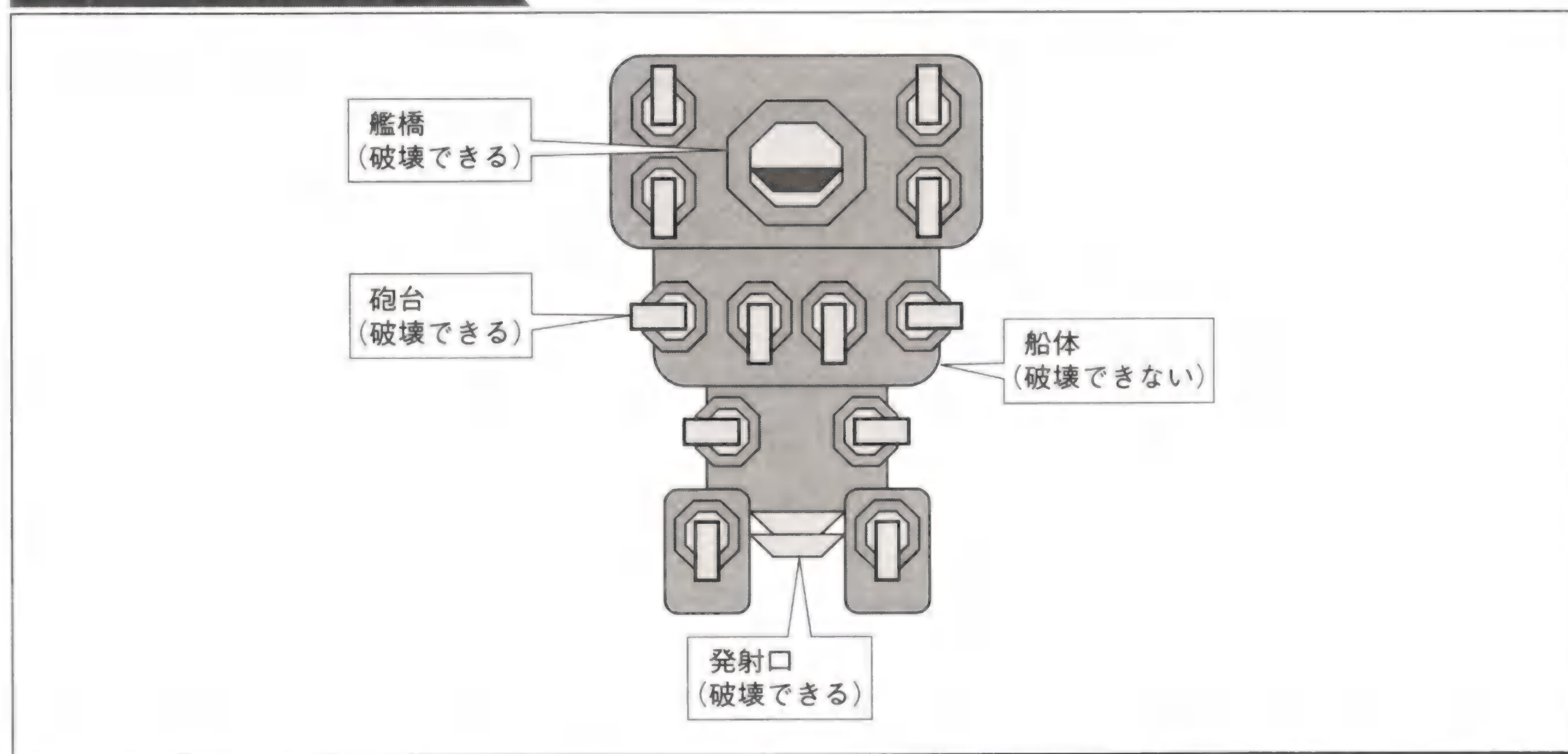
ボスキャラの構造

普通の敵に比べて、ボスキャラの構造はずっと複雑です(Fig. 6-24)。多くのボスキャラには「破壊できる部分」と「破壊できない部分」があります。また、多数の砲台が設置されているときには、個々の砲台を別々に破壊していくことができます。

このように複雑な構造をしたボスキャラを作る方法の1つは、ボスキャラを構成する各部分を「別々の敵」として考えることです(Fig. 6-25)。さまざまな性質を持った多数の敵をボスキャラの形に配置し、一体化させて動かすことによって、あたかも1つのボスキャラであるかのように見せます。ボスキャラを構成するそれぞれの敵について別々に当たり判定処理を行えば、パーツ単位で破壊することも簡単です。

複数の敵を使ってボスを構成しておけば、「特定の部分を破壊したらボス全体が破壊される」といった処理も可能です(Fig. 6-26)。これは「ゼビウス(→ P. 329)」の「アンドアジェネシス」などに見られます。「アンドアジェネシス」の場合には、中心にある「コア」を破壊すると、「アンドアジェネシス」全体を1発で沈めることができます。同様に、「特定のパーツを破壊したらボスキャラの攻撃が次の段階に進む」といった演出も実現可能です。

List 6-11はボスキャラの構造に関するプログラムです。ボスキャラを表す構造体のなかには、敵を表す構造体が複数入っています。

Fig. 6-24 ボスキャラの構造

サンプル

ボス →P. 321

Fig. 6-25 複数の敵を使ってボスキャラを構成する

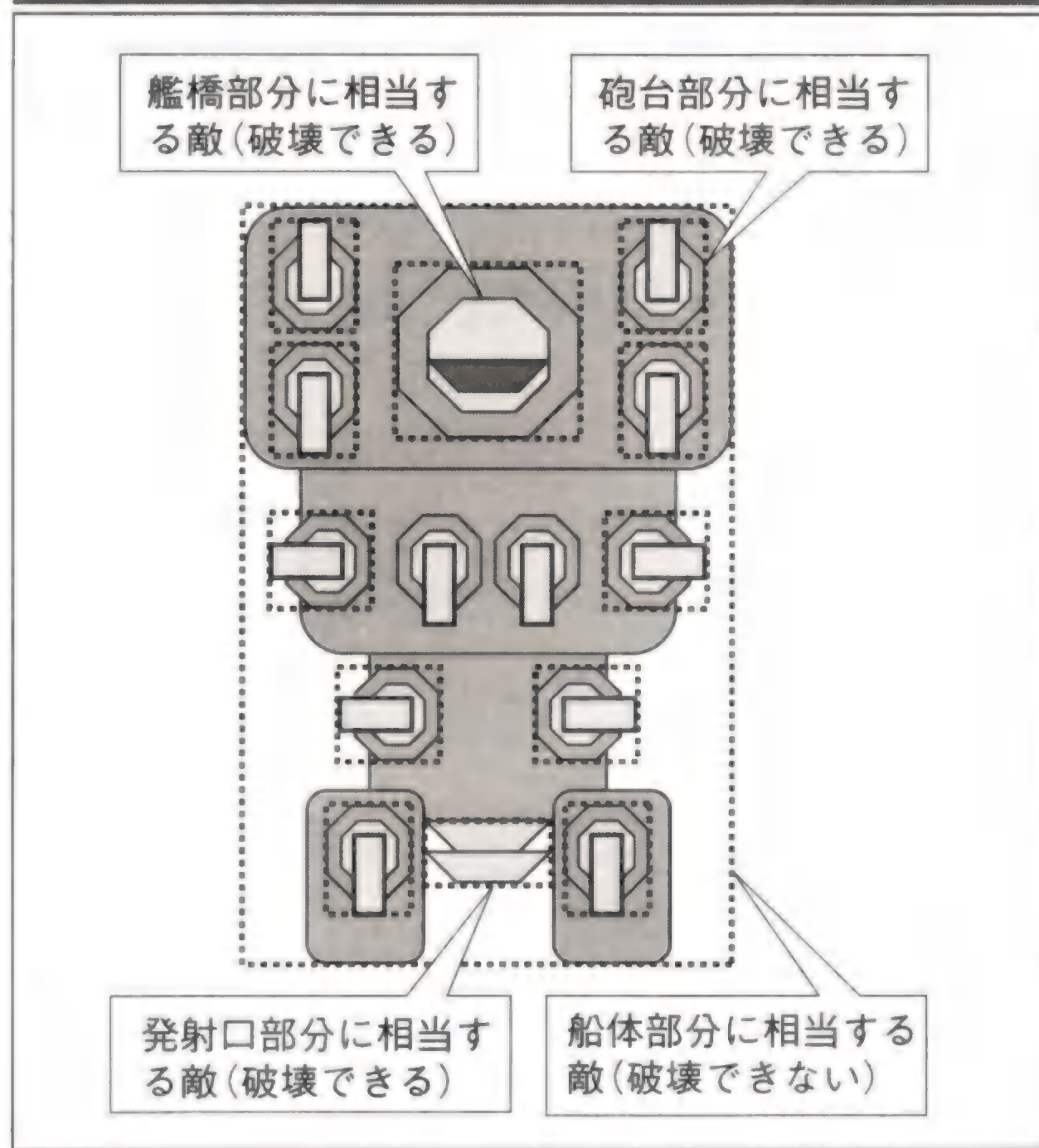
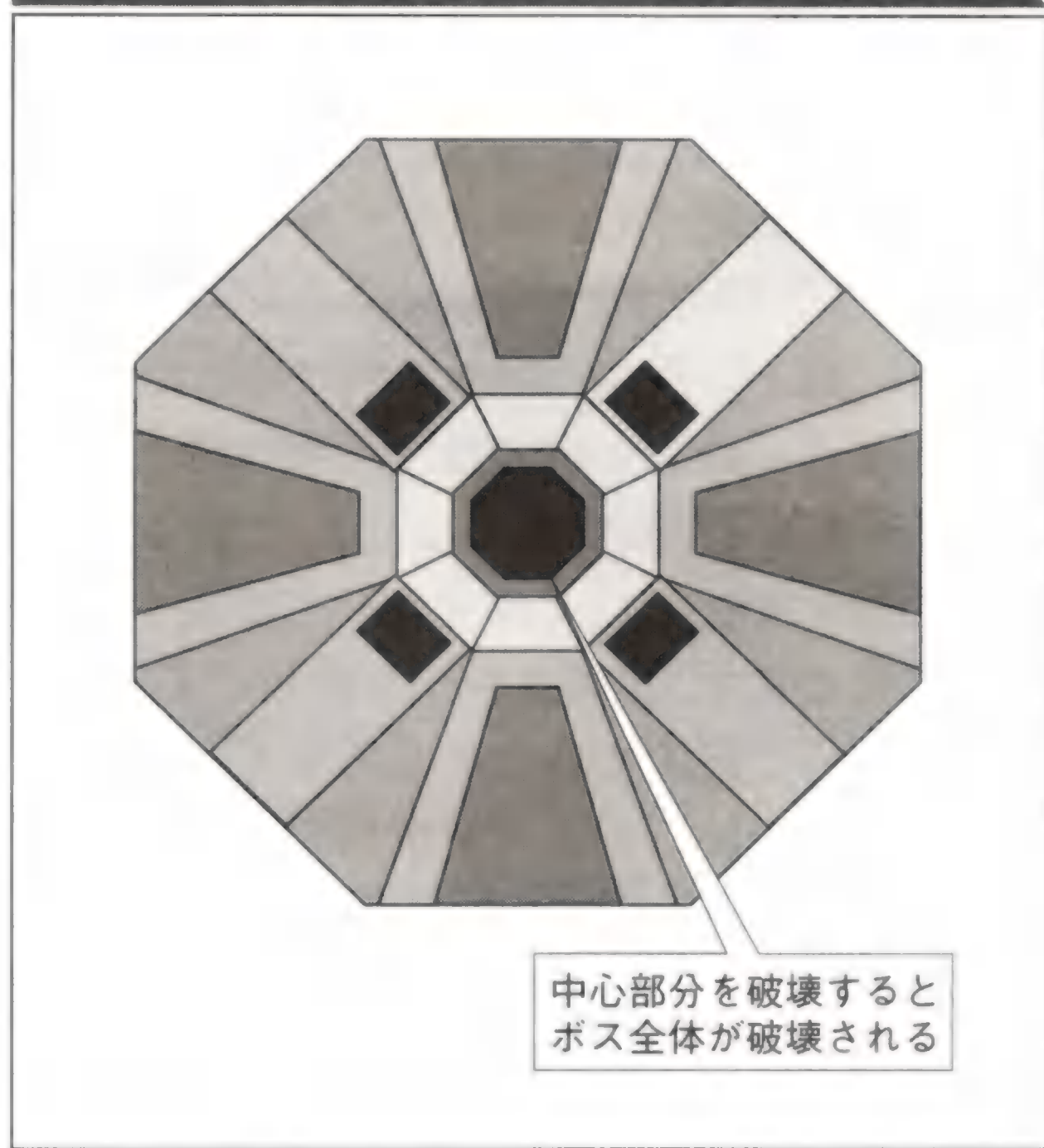



Fig. 6-26 特定部分を破壊するとボスキャラ全体が破壊される

List 6-11 ボスキャラの構造

```
// 敵を表す構造体
typedef struct {
    float X, Y;  // 敵の座標
} ENEMY;

// ボスキャラ（複雑な構造の敵）を表す構造体
#define MAX_PART 32
typedef struct {
    float X, Y;                // ボスキャラの座標
    ENEMY* Part[MAX_PART];     // ボスキャラを構成するパーツ（敵）
    int NumPart;               // パーツ（敵）の数
    bool Fatal[MAX_PART];      // 致命的なパーツかどうか
} COMPLEX_ENEMY;

// ボスキャラの動き：
// ボスキャラを構成するすべての敵について順に処理を行う。
void MoveComplexEnemy(COMPLEX_ENEMY* ce) {
    for (int i=0; i<ce->NumPart; i++) {

        // 敵の行動：
        // 具体的な処理はMoveEnemy関数で行うとする。
        MoveEnemy(ce->Part[i]);

        // 敵の描画：
```

```
        // すべてのパーツを一体化させて動かすために、
        // パーツの座標をボスキャラ座標からの相対位置として使う。
        // ボスキャラ全体を動かすにはce->Xとce->Yを変更する。
        // 具体的な処理はDrawEnemy関数で行うとする。
        DrawEnemy(ce->Part[i],
            ce->Part[i]->X+ce->X, ce->Part[i]->Y+ce->Y);

        // 敵の破壊:
        // 判定と破壊の具体的な処理は、
        // Destroyed、DeleteEnemyの各関数で行うとする。
        if (Destroyed(ce->Part[i])) {
            DeleteEnemy(ce->Part[i]);

            // ボスキャラ全体の破壊:
            // 破壊されたパーツが致命的だったときには、
            // ボス全体を破壊する。
            if (ce->Fatal[i]) DeleteEnemy(ce);
        }
    }
}
```

ボスキャラの行動

ボスキャラにはさまざまな行動パターンがあります。昔のゲームはボスキャラの動きも単純でしたが、最近のゲームに出てくるボスキャラは、数種類の攻撃パターンを使い分けてくるのが普通です (Fig. 6-27)。

複数種類の攻撃パターンを切り替えるには、ボスキャラを「攻撃パターン0」「攻撃パターン1」「攻撃パターン2」「待機」などのいくつかの状態に分けて管理します (Fig. 6-28)。ある状態の行動を終えた時点で別の状態に移ることによって、さまざまな攻撃を出します。

状態遷移のタイミングや次に遷移する状態の選び方は、ボスキャラによってさまざまです。たとえば「攻撃0」→「攻撃1」→「攻撃2」→「待機」のように順番に遷移しても、いずれかのパターンをランダムに出してもよいでしょう。賢い動きをするボスキャラにするのなら、自機の位置などから判断してもっとも効果的な攻撃パターンを出すという方法もあります。ただ、シューティングゲームの目的は「自機を倒す」ことではなく「適度なスリルを味あわせる」ことなので、ボスキャラの動きをあえて規則的にするというのも1つの方法です。

List 6-12はボスキャラの行動に関してまとめたものです。このプログラムでは「攻撃0」「攻撃1」「攻撃2」「待機」という順番を繰り返します。

Fig. 6-27 ボスキャラの行動

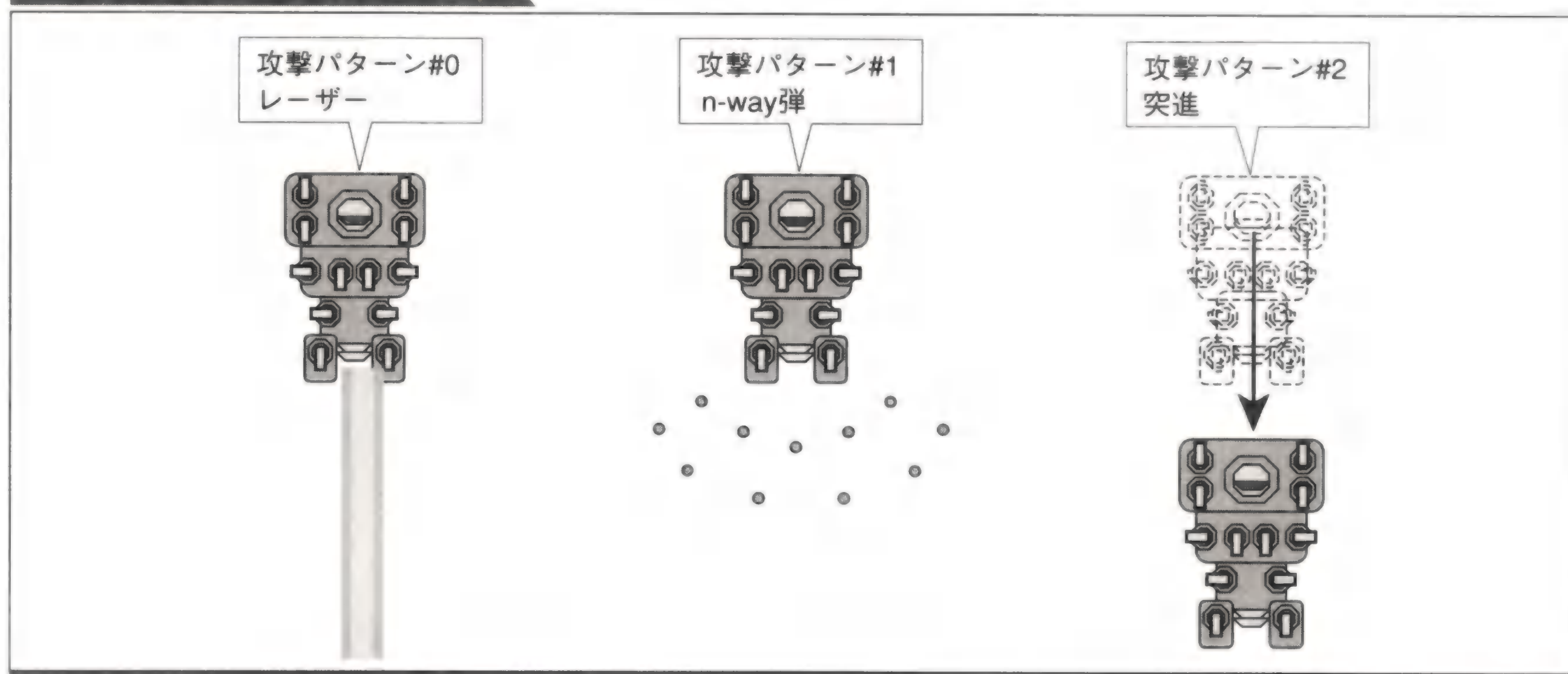
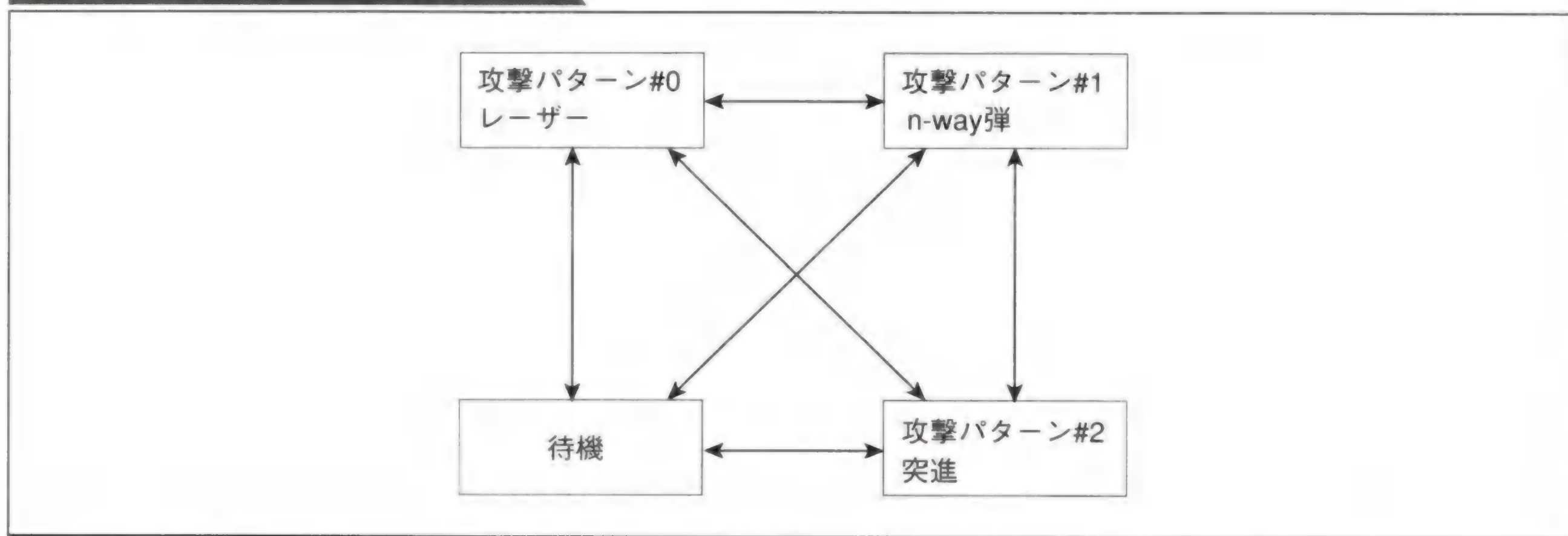



Fig. 6-28 ボスキャラの状態遷移

List 6-12 ボスキャラの行動

```
/ ボスキャラの状態：
// 攻撃0、攻撃1、攻撃2、待機
typedef enum {
    ATTACK0, ATTACK1, ATTACK2, WAIT
} BOSS_STATE;

// 各状態の制限時間
int Time[]={30, 40, 25, 20};

// ボスキャラの行動
void MoveBoss(
    BOSS_STATE& state,  // 現在の状態
    int timer           // タイマー
```

```
) {
    // 攻撃：
    // 状態に応じた攻撃を行う。待機のときには何もしない。
    // 攻撃の具体的な処理は、
    // Attack0、Attack1、Attack2の各関数で行うとする。
    switch (state) {
        case ATTACK0: Attack0(); break;
        case ATTACK1: Attack1(); break;
        case ATTACK2: Attack2(); break;
    }

    // 状態遷移：
    // タイマーを加算し、
    // 各状態の制限時間を超えたら次の状態に移行する。
    timer++;
    if (timer>Time[state]) {
        state=(Boss.STAGE)((stage+1)%4);
        timer=0;
    }
}
```

ボスキャラの分離と合体

ボスキャラのなかには分離攻撃をしかけてくるものがあります（Fig. 6-29）。分離後のパーツはそれぞれ独立して行動します。しばらく分離攻撃をしたあとに、また合体して元の姿に戻ることもあります。

「ボスキャラの構造」（→ P. 237）で解説したように、ボスキャラをいくつかの別々の敵から構成しておけば、分離や合体を実現することは難しくありません（Fig. 6-30）。分離するパーツはあらかじめ別々の敵にしておき、合体時には全パーツを一体化して動かし、分離時には各パーツを単独で動かします。

分離するパーツ自体が複雑な形をしているときには、各パーツをボスキャラと同じ作り方で構成します。つまり、複数の小さなボスキャラを集めて1つの大きなボスキャラにするということです。

分離と合体に関する処理はList 6-13のようになります。ここでは分離中と合体中について、それぞれボスキャラを動かす関数を用意しました。

Fig. 6-29 分離するボスキャラ

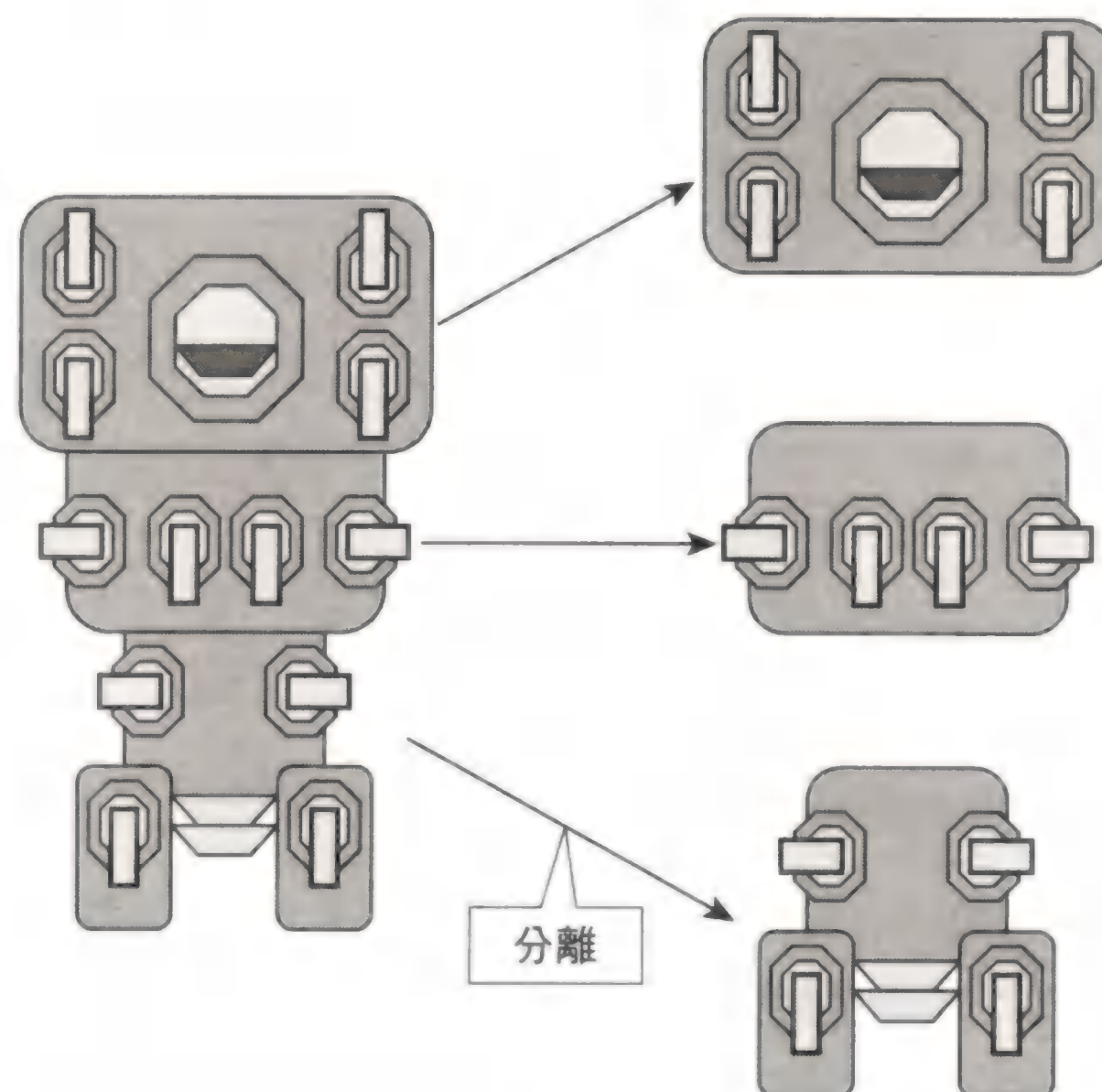
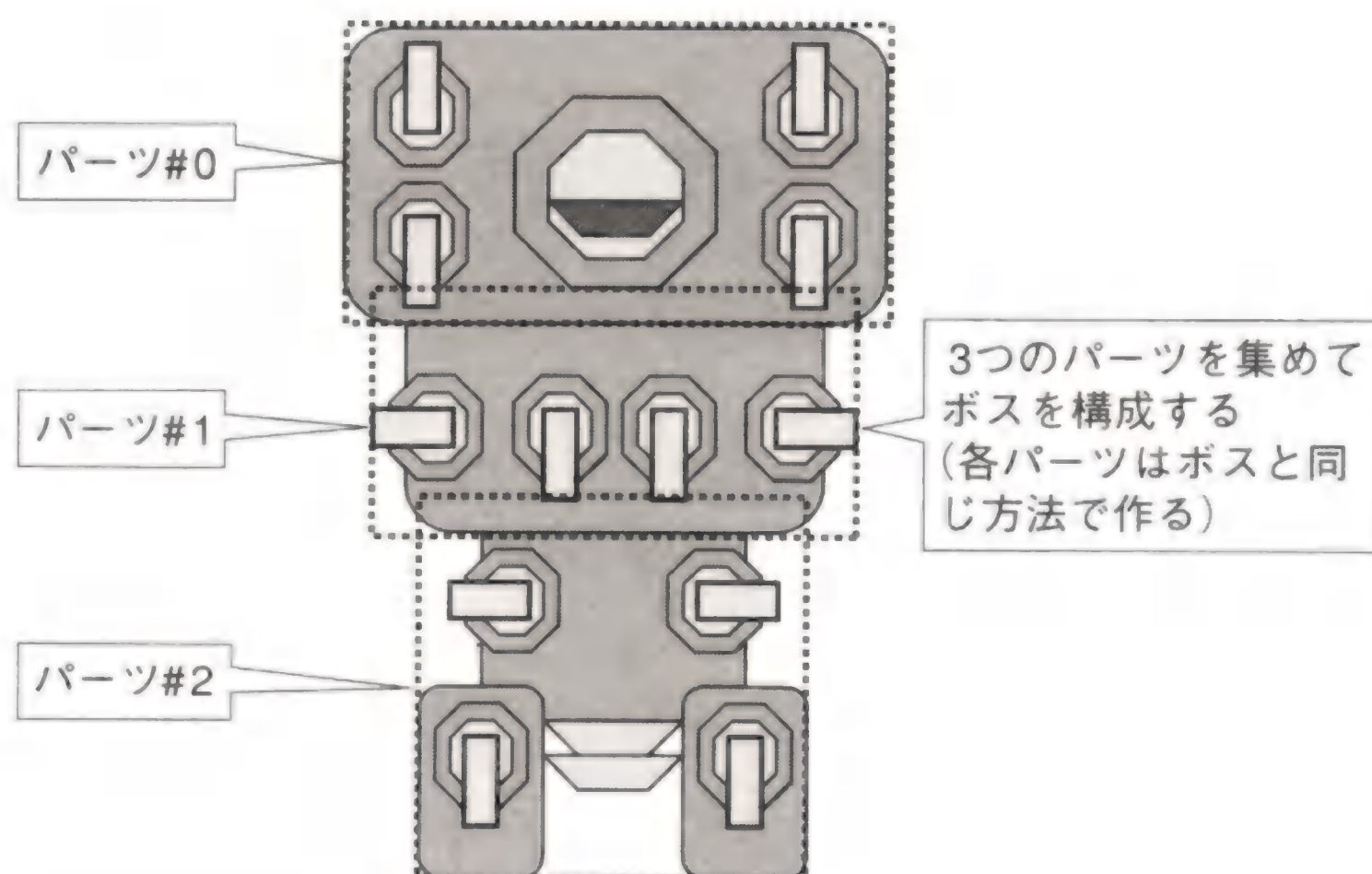



Fig. 6-30 分離と合体の実現方法

List 6-13 ボスキャラの分離と合体

```
// ボスキャラを表す構造体
#define MAX_PART 32
typedef struct BOSS_STRUCT {
    float X, Y;             // ボスキャラの座標
    struct BOSS_STRUCT*
        Part[MAX_PART];     // ボスキャラを構成するパーツ(ボスキャラ)
    int NumPart;            // パーツの数
    float PX[MAX_PART],
```

```
        PY[MAX_PART];     // パーツの相対座標
} BOSS;

// 分離中の動き：
// 各パーツを独立に動かす。
// 移動の具体的な処理はMoveBoss関数で行うとする。
void MoveSeparatedBoss(BOSS* boss) {
    for (int i=0; i<boss->NumPart; i++) {
        MoveBoss(boss->Part[i]);
    }
}

// 合体中の動き：
// 全パーツが一体化して動くように、
// 中心となる座標に対して固定の位置にパーツを配置する。
// 移動の具体的な処理はMoveBoss関数で行うとする。
void MoveUnitedBoss(BOSS* boss) {
    MoveBoss(boss);
    for (int i=0; i<boss->NumPart; i++) {
        boss->Part[i]->X=boss->X+boss->PX[i];
        boss->Part[i]->Y=boss->Y+boss->PY[i];
    }
}
```

デモプログラム
Stage#80　ボス

ボスキャラの変形

ボスキャラのなかには形態を変えながら攻撃をしかけてくるものがあります（Fig. 6-31）。一般にボスの形態が変わると、攻撃方法や弱点も変わります。ボスキャラの形態が変わるタイミングはさまざまですが、時間とともに勝手に形を変えるものと、特定のパーツを破壊したときに次の形態に移るものとがあります。

変形の実現方法は、「ボスキャラの分離と合体」（→ P. 241）の実現方法に似ています（Fig. 6-32）。

Fig. 6-31 変形するボスキャラ

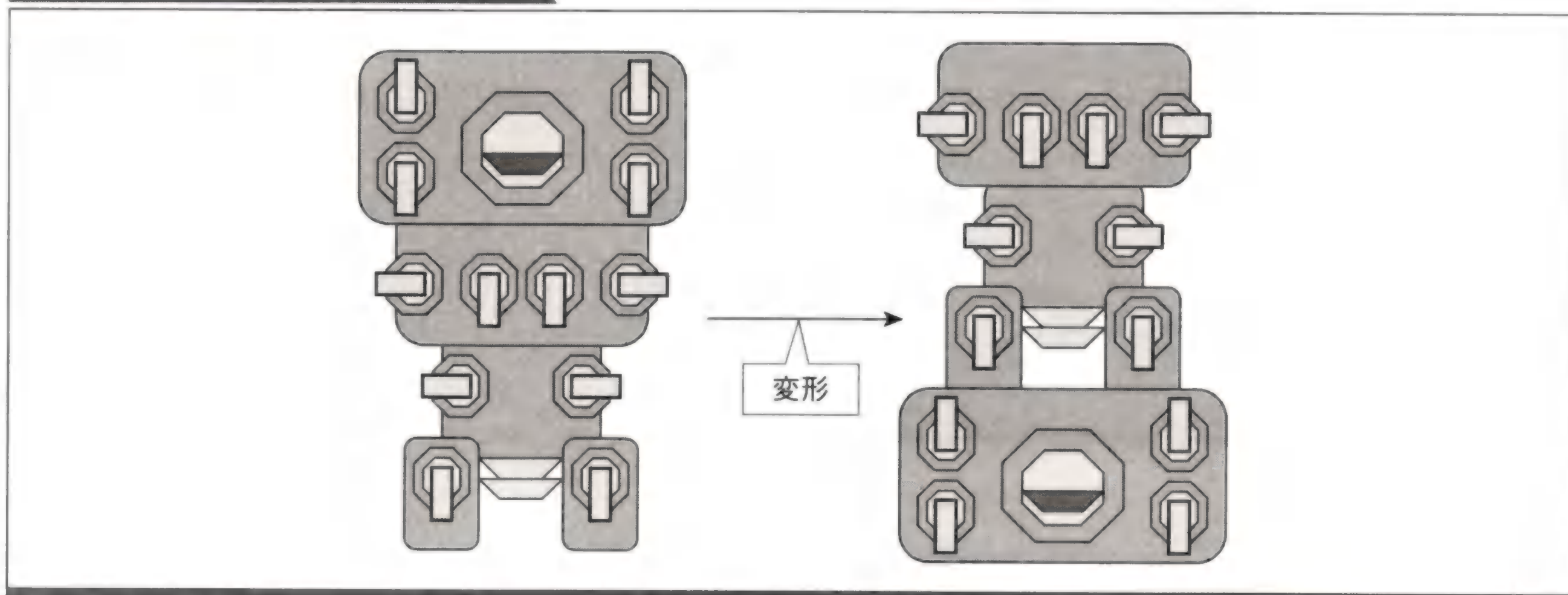
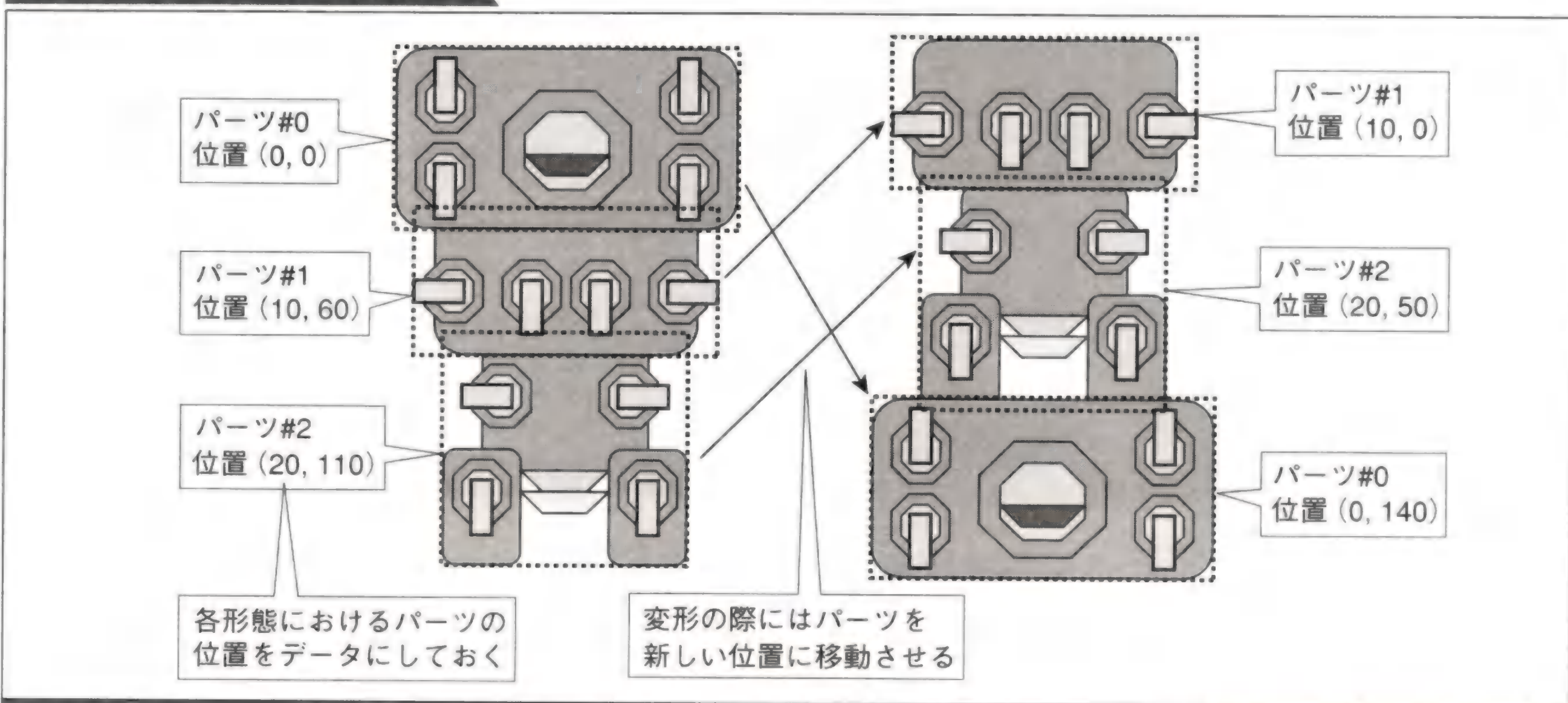



Fig. 6-32 変形の実現方法

ポイントは、「ボスキャラの構造」(→ P. 237)で解説したように、ボスキャラをいくつかの別々のパーツ(敵)から構成しておくことです(グラフィックそのものが変わって別の姿になるものもあります)。そして、各形態におけるパーツの位置をデータにしておきます。変形する際にはこのデータに従って、各パーツを新しい位置へ移動させます。

List 6-14は変形に関する処理をまとめたプログラムです。もっと複雑な変形を実現したいときには、変形の際にパーツを新たに生成したり、不要なパーツを削除したりといった処理を加えてもよいでしょう。

List 6-14 ボスキャラの変形

```
// ボスキャラを表す構造体
#define MAX_PART 32
#define MAX_FORM 8
typedef struct BOSS_STRUCT {
    float X, Y;                      // ボスキャラの座標
    struct BOSS_STRUCT*
        Part[MAX_PART];              // ボスキャラを構成するパーツ(ボスキャラ)
    int NumPart;                     // パーツの数
    float
        PX[MAX_PART][MAX_FORM],      // パーツの相対座標
        PY[MAX_PART][MAX_FORM];      // [パーツ番号][形態番号]
} BOSS;

// 変形:
// 各パーツを変形前の位置から変形後の位置へ移動させる。
// 変形にかかる時間とタイマーとの比率から、
// 移動途中の位置を計算する。
#define TRANSFORM_TIME 180
void TransformBoss(
    BOSS* boss,  // ボスの構造体へのポインタ
    int from,    // 変形前の形態番号
    int to,      // 変形後の形態番号
    int& timer   // タイマー
) {
    float ratio=(float)timer/TRANSFORM_TIME;
    for (int i=0; i<boss->NumPart; i++) {
        boss->Part[i]->X=
            boss->PX[i][from]*(1-ratio)+
            boss->PX[i][to]*ratio;
        boss->Part[i]->Y=
            boss->PY[i][from]*(1-ratio)+
            boss->PY[i][to]*ratio;
    }
}
```

触手

伸びたり縮んだりする腕状の物体です（Fig. 6-33）。触手が単体で敵になる場合もあれば、触手を何本も生やした大型の敵が出てくる場合もあります。また、生物をモチーフにしたステージでは、床や壁面に触手が配置されていることがよくあります。

触手の表現方法はゲームによって異なりますが、触手の一部分となる小さなパーツを多数並べて表現するのが一般的です（Fig. 6-34）。触手がグニャグニャ動いても不自然にならないように、パーツは円形または円に近い形をしています。

Fig. 6-33 触手

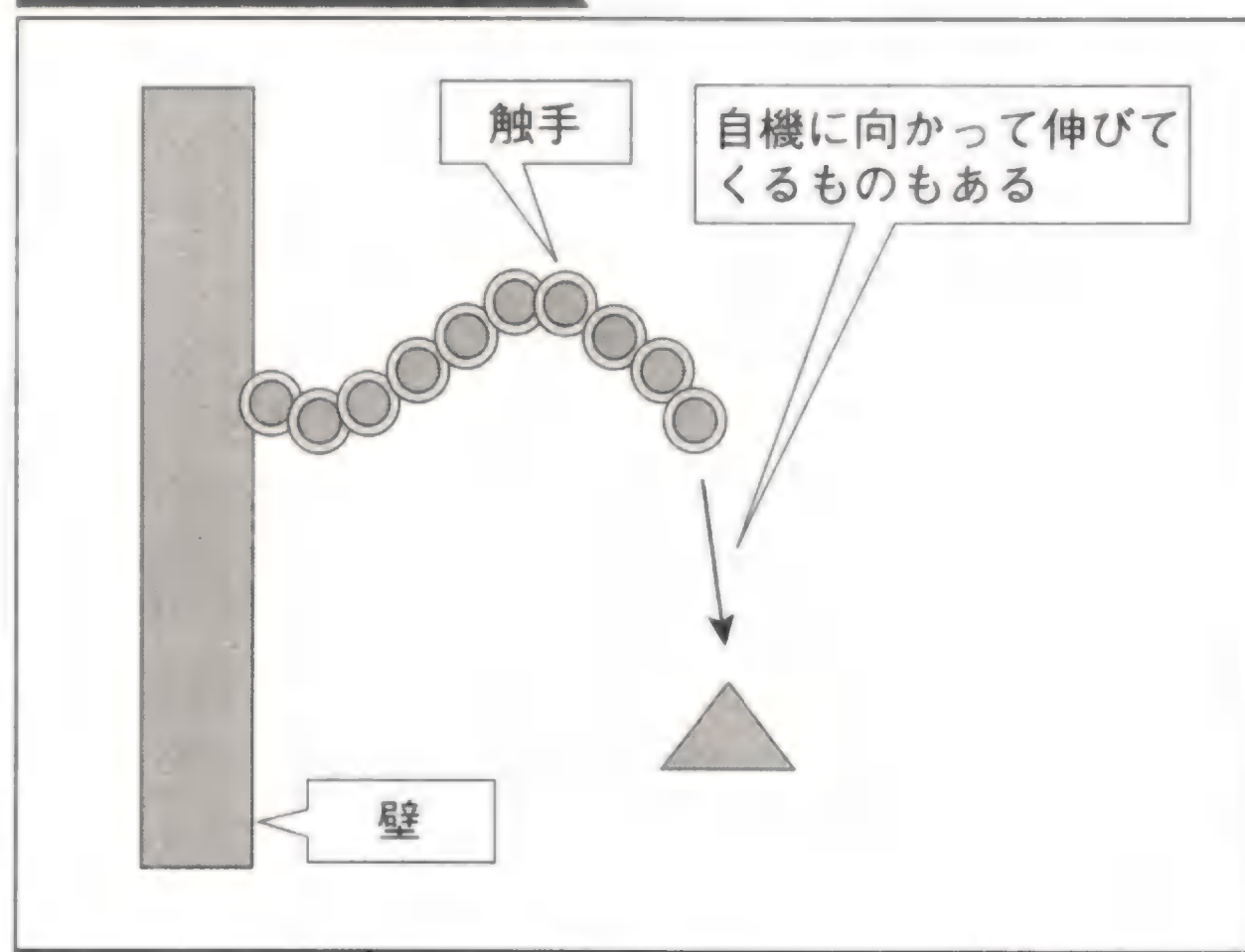
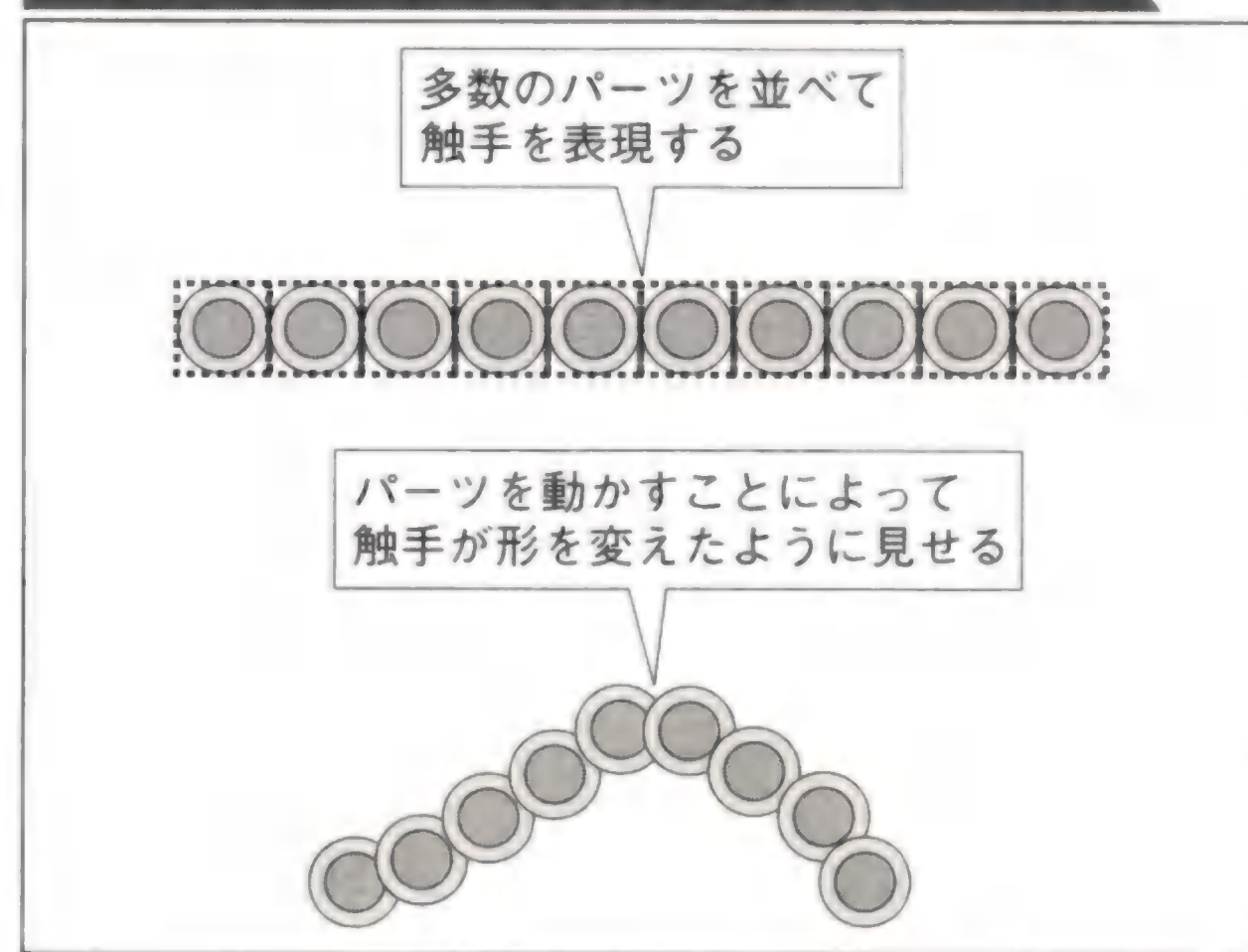



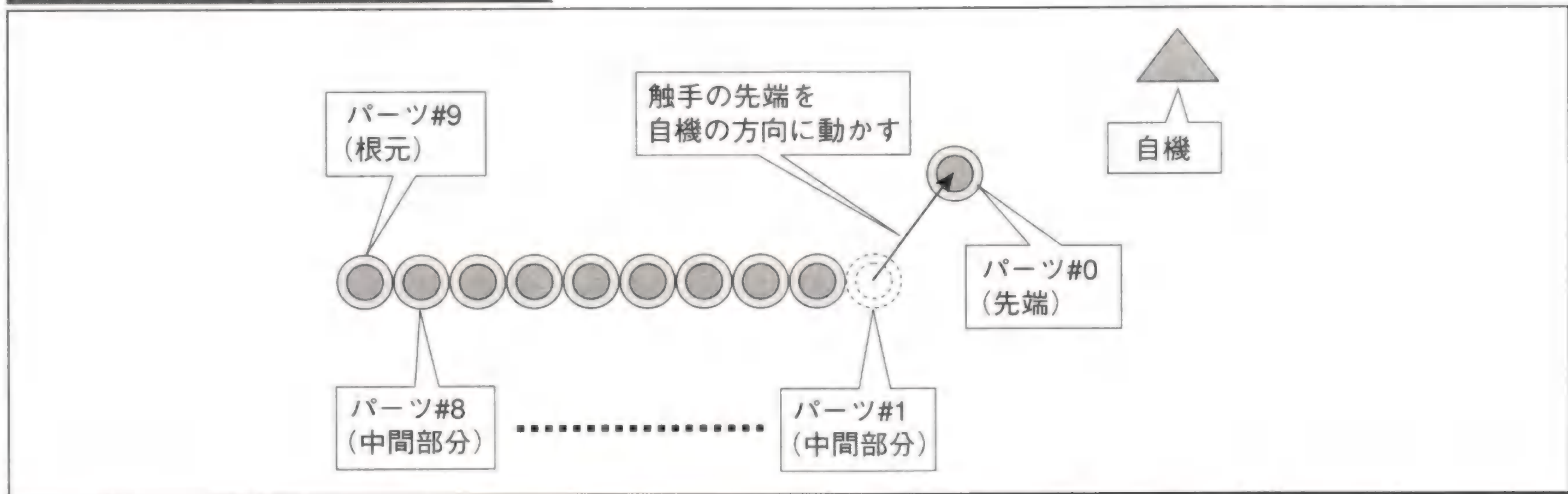
Fig. 6-34 パーツを並べて触手を表現する

現在は、シューティングゲームでも3Dグラフィックを使うケースが増えたので、触手の表現方法も多様化しました。3Dを使えばキャラクターの形を比較的自由に変形させることができるので、触手を1つのキャラクターとして作ってしまい、あとはそれを変形させて動かすということも可能です。しかし現在においても、動作が軽いゲームを作りたいときや、2Dグラフィックを使う必要があるとき、あるいは少しレトロな雰囲気のゲームを作りたいときには、パーツを並べて触手を表現する手法が役に立ちます。

触手を動かすにはさまざまな方法がありますが、ここでは計算が簡単な方法を1つ解説します。おおまかな手順は次のようになります。

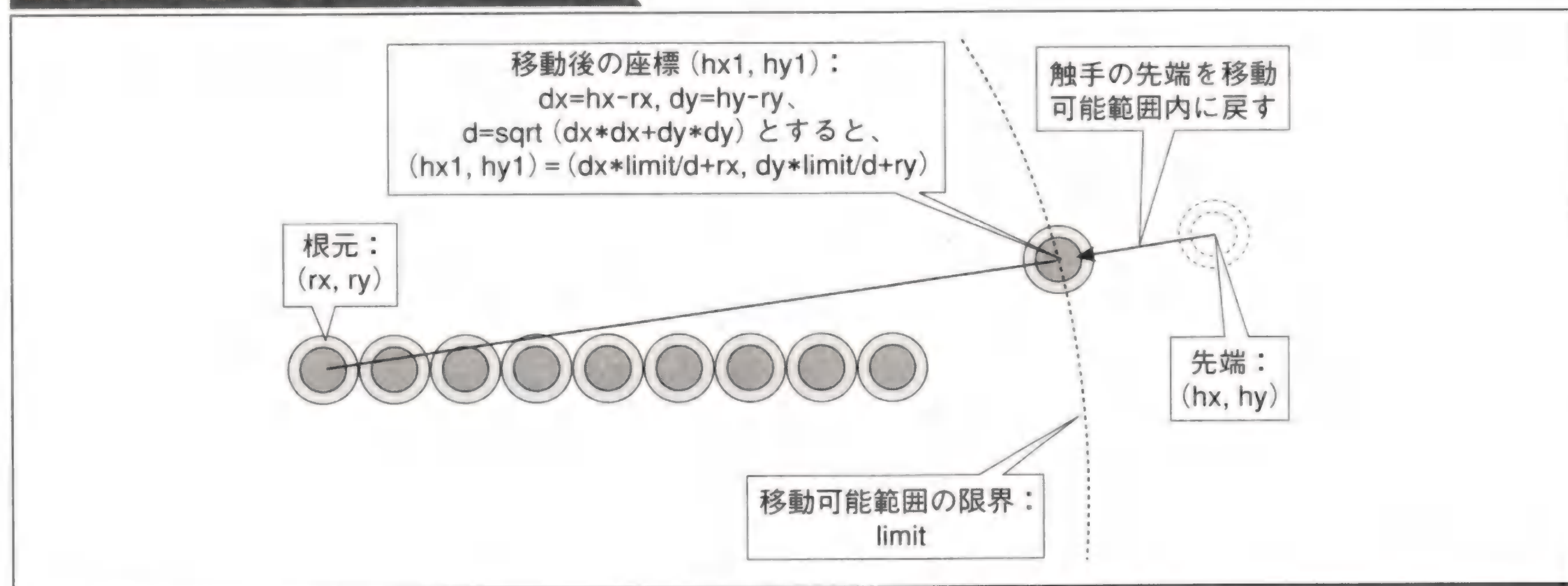
①触手の先端を動かす
②触手の中間部分（先端と根元以外）を動かす
③必要ならば②を何度か繰り返す

■ 触手の先端を動かす

まずは触手の先端を動かします (Fig. 6-35)。ここでは触手を10個のパーツで構成することにして、先端を「パーツ#0」、根元を「パーツ#9」としました。先端を動かす方法はさまざまですが、ここでは「誘導弾」(→ P. 39) で旋回角度の上限を設けない場合と同じように、単純に自機を追いかけることにします。

Fig. 6-35 触手の先端を動かす

触手の先端が移動する速さは一定とします。移動方向は自機と先端との位置関係によって決まります。ここでは計算を簡単にするため、X方向とY方向を別々に処理することにしました。たとえば自機が右上にある場合には、X方向の移動速度を+1、Y方向の移動速度を−1します。

必要ならば、触手の移動範囲を制限することもできます (Fig. 6-36)。触手の先端と根元との距離が一定値を超えていたら、移動可能範囲内に戻せばよいのです。

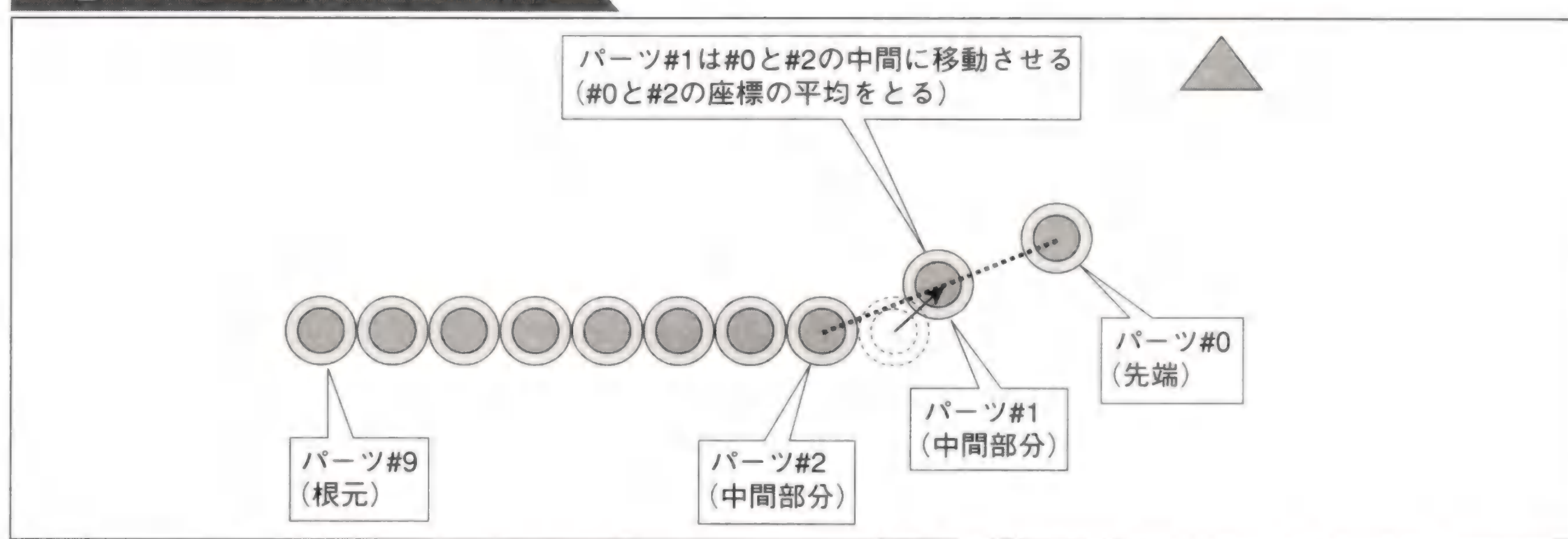
Fig. 6-36 触手の移動範囲を制限する

移動前の先端の座標を（hx, hy）、根元の座標を（rx, ry）、移動可能範囲の限界（根元からの距離）をlimitとすると、移動後の先端の座標（hx1, hy1）は次のように求められます。

```
dx=hx-rx
dy=hy-ry
d=sqrt (dx*dx+dy*dy)
(hx1, hy1) = (dx*limit/d+rx, dy*limit/d+ry)
```

ほぼ同じ方法で、触手を一定の長さより縮めないようにすることもできます。

■ 触手の中間部分を動かす

次に触手の中間部分を動かします。中間部分は、感覚的に表現すれば「触手の先端と根元の両方から引っ張られている」わけです。これを正確に計算するには運動方程式を解かなければならないのですが、ここでは非常におおざっぱに「中間部分の座標は隣接する部分の座標の平均値とする」という計算方法を使います（Fig. 6-37）。

Fig. 6-37 中間部分の座標を求める

正確な計算ではありませんが、隣接部分の座標の平均値をとることによって、中間部分の座標は両端の座標の影響を受けることになります。結果として、あたかもヒモの先端をつかんで引っ張ったときのような動きを触手にさせることができます。

中間を動かす処理の繰り返しは必ずしも行わなくてもよいのですが、何度か繰り返すと触手の動きがなめらかになることがあります。この計算方法では先端の動きが根元に伝わるまでに時間がかかるので、何度か計算を繰り返すことによって、動きが伝わるのを速くするのです。適度に繰り返しをすると、先端だけが大きく動いてしまうのではなく、触手全体がなめらかに動くようになります。

List 6-15は触手の処理をまとめたプログラムです。移動範囲を制限する必要がないときには、移動範囲の制限に関する処理が丸ごと不要になるので、非常に簡単なプログラムになります。

サンプル

触手 → P. 321

List 6-15 触手

```
#include <math.h>

void MoveTentacle(
    float x[], float y[],  // 各パーツの座標
    int num_part,          // パーツの数
    float tx, float ty,    // 目標の座標
    float v,               // 先端が移動する速さ
    float limit,           // 先端と根元の最大距離
    int num_loop           // 計算の繰り返し回数
) {
    // 先端の座標と根元の座標
    float hx=x[0], hy=y[0];
    float rx=x[num_part-1], ry=y[num_part-1];

    // 目標の方向に先端を動かす
    hx+=(tx>=hx+v ? v : (tx<=hx-v ? -v : 0));
    hy+=(ty>=hy+v ? v : (ty<=hy-v ? -v : 0));

    // 先端の移動範囲を制限する：
    // 先端と根元との距離が限界値を超えていたら、移動可能範囲内に戻す。
    float dx=hx-rx, dy=hy-ry;
    float d=sqrt(dx*dx+dy*dy);
    if (d>limit) {
        hx=dx*limit/d+rx;
        hy=dy*limit/d+ry;
    }

    // 中間部分の座標を計算する：
    // 隣接部分の座標の平均をとる。計算は先端から根元に向かって行う。
    // 必要に応じて計算を複数回繰り返す。
    x[0]=hx; y[0]=hy;
    for (int l=0; l<num_loop; l++) {
        for (int i=1; i<num_part-1; i++) {
            x[i]=(x[i-1]+x[i+1])/2;
            y[i]=(y[i-1]+y[i+1])/2;
        }
    }
}
```

多関節

多くの関節を持つ物体です（Fig. 6-38）。「触手」（→ P. 246）とよく似ていますが、触手が伸び縮みするのに対して、ここでは伸び縮みしないものについて解説します。ゲームによっては「触手」と「多関節」という言葉を同じ意味で使うこともありますが、ここでは便宜上、伸び縮みするものを「触手」、伸び縮みしないものを「多関節」と呼ぶことにします。

実際のゲームでは、触手や触角のようなものを多関節で表現することもあれば、昆虫や甲殻類などの脚に多関節を使うこともあります（Fig. 6-39）。ゲームによっては地形から多関節キャラ（多くの関節を持ったキャラクター）が生えていたり、多関節の先端に砲台がついていたりします。

Fig. 6-38 多関節

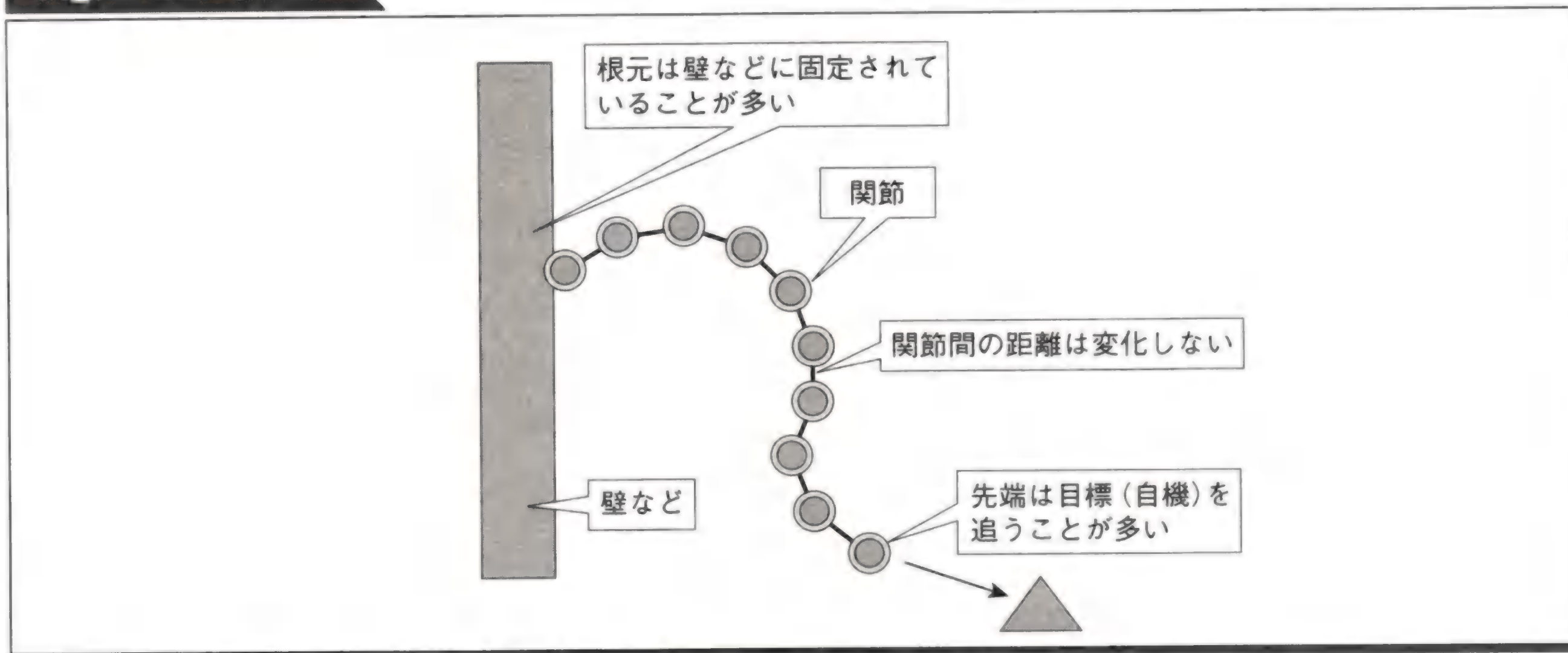
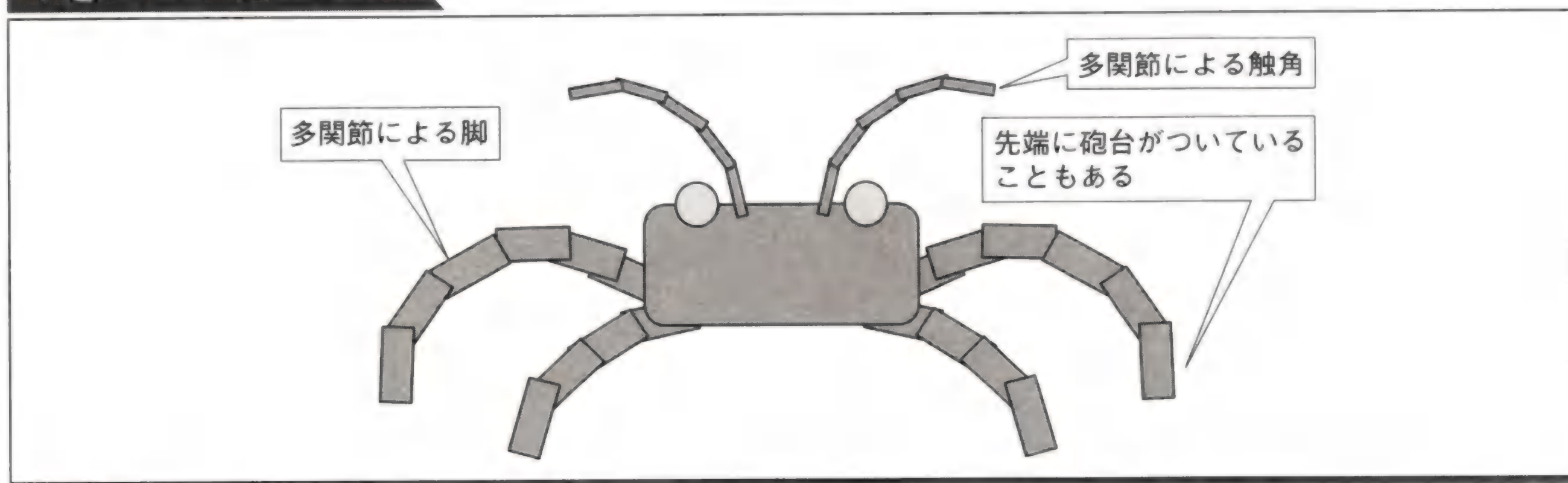
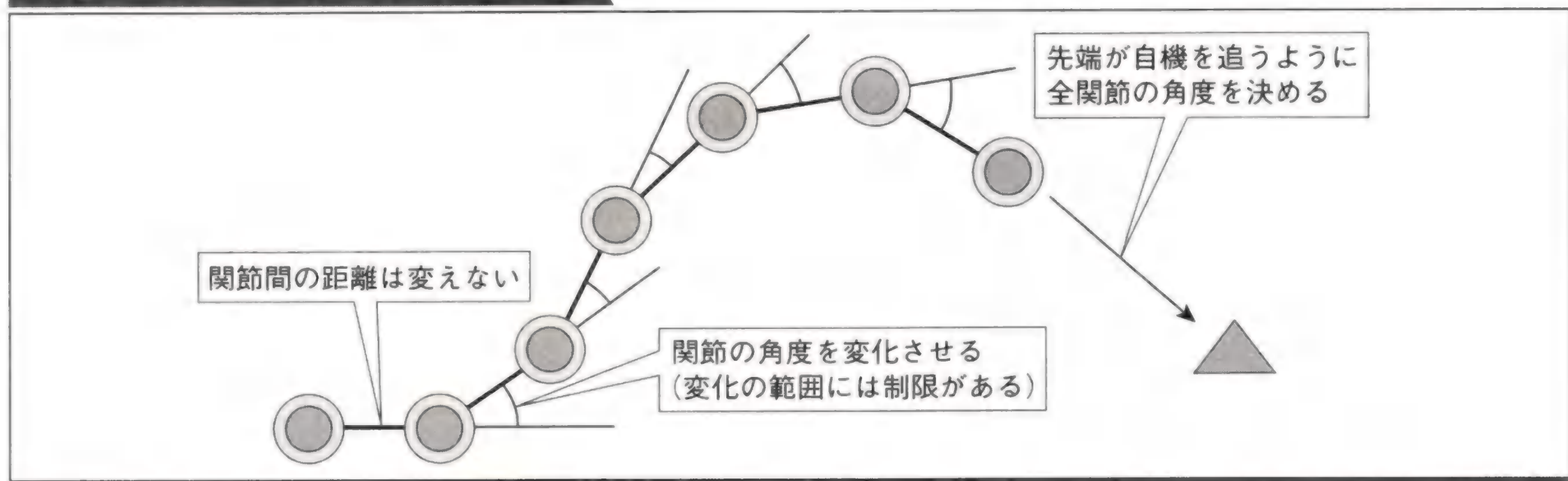



Fig. 6-39 多関節の例

多関節のポイントは「伸び縮みしない」ということです。つまり、隣接する関節間の距離は変化しません。「触手」で解説したアルゴリズムでは関節間の距離が変わってしまうので、別のアルゴリズムを使います。

多関節を動かすには、各関節の角度だけを変化させます（Fig. 6-40）。関節間の距離は一定なので、関節の角度が決まれば、全関節の座標を求めることができます。

Fig. 6-40　関節の角度を変化させる

問題は、各関節の角度をどう決めるかです。ここでは先端が目標（自機など）を追うように関節の角度を決める方法を考えます。これは「インバースキネマティクス（IK）」と呼ばれる技法の一種です。

インバースキネマティクスは主に3Dグラフィックの分野で使われる技術で、人体モデルなどの動きを決める際に用います。たとえば、人間がものをつかむ動作を3Dグラフィックで作ろうとしたときには、次のことを決めなければなりません。

- **肩関節の角度**
- **肘関節の角度**
- **手首関節の角度**

これらの値を1つひとつ手作業で設定していたのではたいへんですが、高機能な3DCGソフトウェアでは「手を目標物のところまでドラッグすれば、各関節の角度は自動的に調整される」という方式になっています。このときに「手を特定の位置に移動させることができるように各関節の角度を決める」ための方法がインバースキネマティクスです。

話を元に戻して、多関節を動かす方法について解説します。この方法で考える多関節は次のようなものです。

- **伸び縮みしない（関節間の距離は一定）**
- **各関節の回転角度には制限がある（たとえば-10～+10度など）**

人間の腕の関節にも回転角度の限界があるように、多関節の関節にも回転角度の制限を設けたほうが、ゲームとしては面白くなります。関節の曲がる範囲に限界ができるので、その限界をうまく利用して攻撃をよけることができるからです。

さて、多関節を動かす手順はおおまかに次の2段階に分かれています。

- **先端から根元に向かって、各関節の角度を決める**
- **根元から先端に向かって、各関節の位置を求める**

前半は先端から根元、後半は根元から先端に向かって計算することがポイントです。直観的に説明すれば、これには次のような理由があります。

- **角度を決めるときには、なるべく先端に近い部分だけを動かせばすむように、先端から根元に向かって動かす**
- **位置を決めるときには、根元に近い関節が回るとそこから先の関節は全部いっしょに回るので、根元から計算したほうが効率がよい**

それでは具体的なアルゴリズムを説明します。手順が多いので、角度を決める前半と、位置を決める後半とに分けて説明します。

■ 多関節のアルゴリズム (角度の決定)

先端を除いたすべての関節について角度を決めます (先端は回らないので)。関節の現在の角度をradとし (Fig. 6-41)、関節が回転する速さ (単位はラジアン) をvradとすると、各関節は次の3通りのうちいずれかの動きをします (Fig. 6-42)。

- **回らない**
- **右に回る (rad+=vrad)**
- **左に回る (rad-=vrad)**

3つの動きのなかからどれを選ぶかは、「どの動きをとれば先端が目標の方向を向くか」を考慮して決めます (Fig. 6-43)。関節から先端に向かうベクトルを3通りの方向 (回らない、右回り、左回り) に回転させてみて、どのベクトルが目標方向にもっとも近いかを調べます。目標方向に近いベクトルを選ぶには、ベクトルの内積を使うのが簡単です。関節から目標に向かうベクトルと、3通りのベクトルとの間で内積をとると、方向がいちばん近いベクトルに関する内積が最大になります。

Fig. 6-41 関節の現在の角度

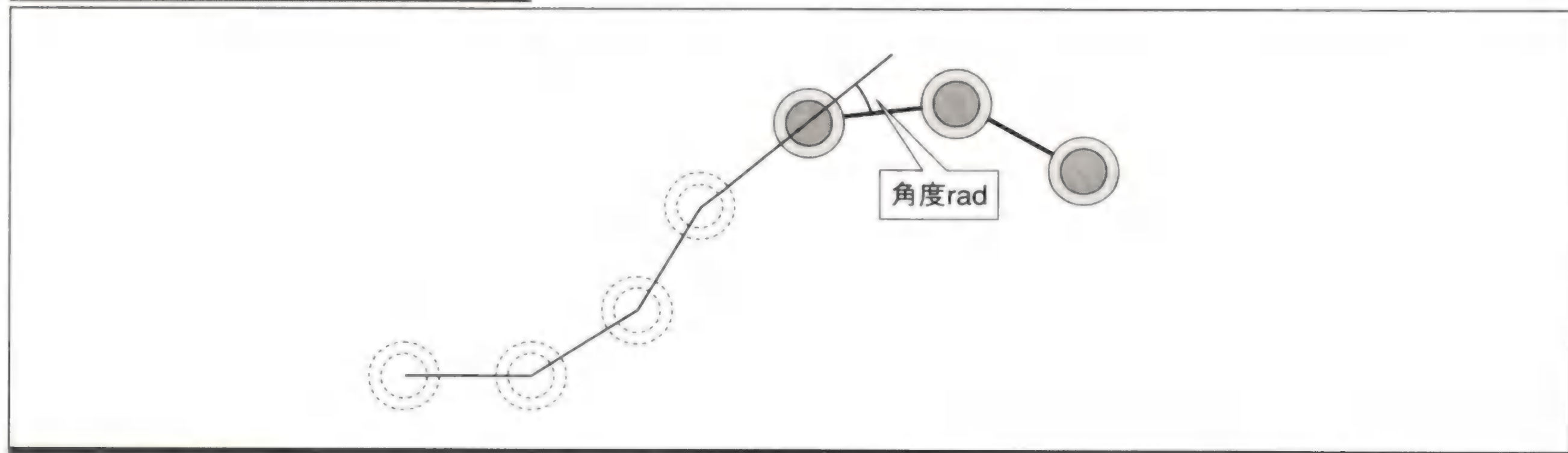



Fig. 6-42 関節の3通りの動き

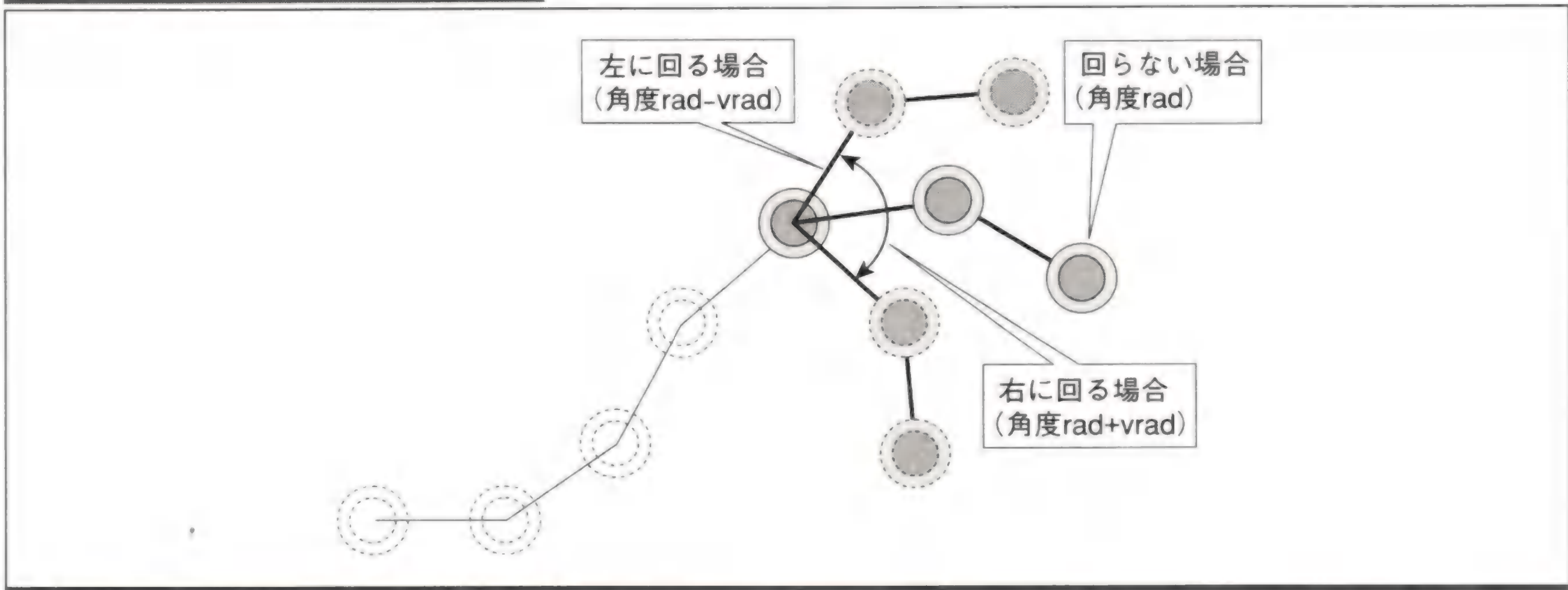
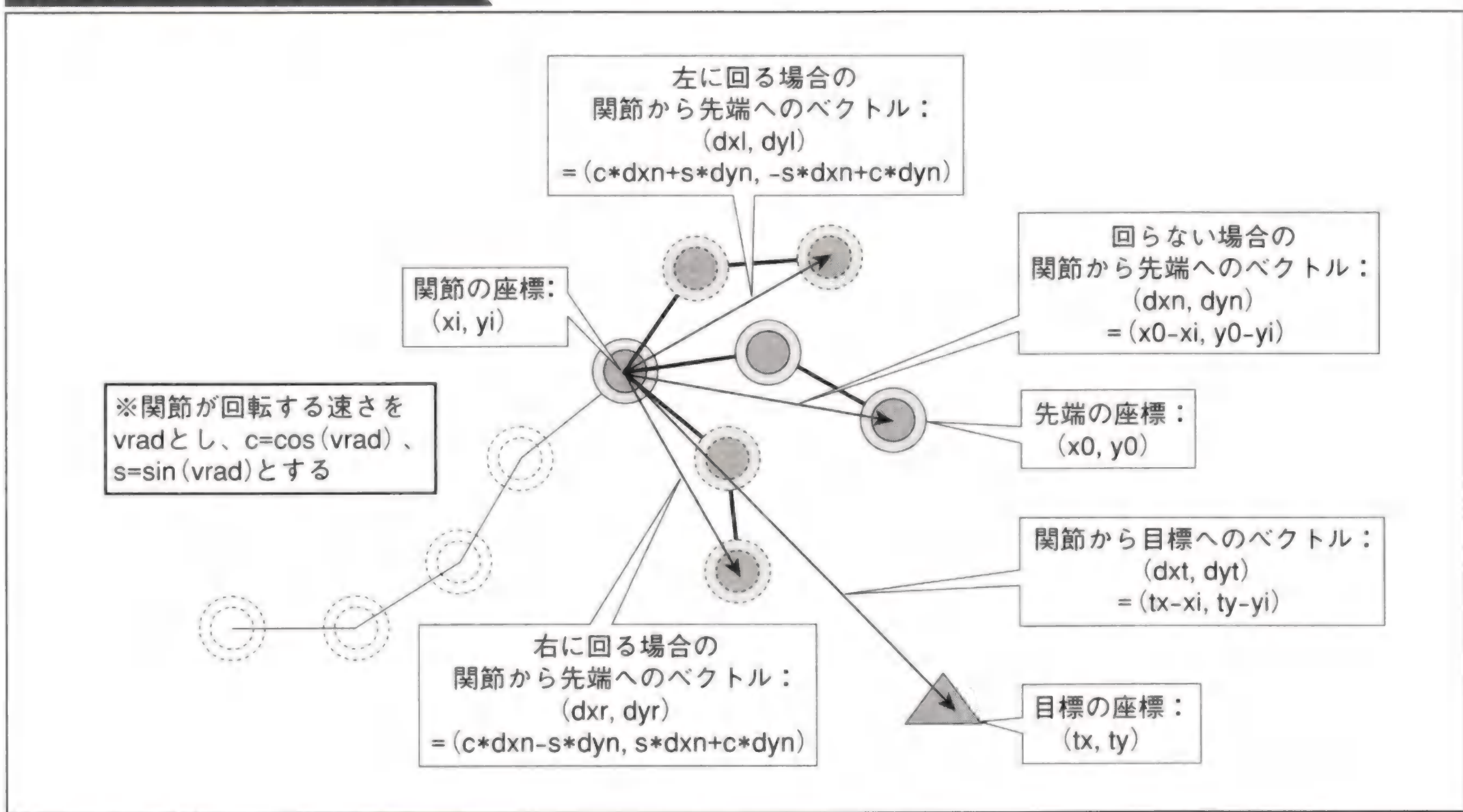



Fig. 6-43 関節の動きを選ぶ

目標の座標を（tx, ty）、関節の座標を（xi, yi）、先端の座標を（x0, y0）とします。関節から目標に向かうベクトルを、

(dxt, dyt) = (tx-xi, ty-yi)

とし、

c=cos (vrad)
s=sin (vrad)

とすると、3通りのベクトル（回らない、右回り、左回り）は次のように計算できます。

◇回らない場合
- **ベクトル：(dxn, dyn) = (x0-xi, y0-yi)**
- **内積　　：dxt*dxn+dyt*dyn**

◇右回りの場合
- **ベクトル：(dxr, dyr) = (c*dxn-s*dyn, s*dxn+c*dyn)**
- **内積　　：dxt*dxr+dyt*dyr**

◇左回りの場合
- **ベクトル：(dxl, dyl) = (c*dxn+s*dyn, -s*dxn+c*dyn)**
- **内積　　：dxt*dxl+dyt*dyl**

求めた3つの内積を比べて、もっとも内積が大きくなる動きを選びます。たとえば右回りの内積が大きければ、その関節は右に回転します。

同じ要領で先端から根元に向かって関節の角度を決めていきます。ポイントは「各関節から先端を見たときに、先端が目標の方向に近づくように回転角度を決める」ということです）。

■ 多関節のアルゴリズム（位置の計算）

全関節の角度が決まったら、今度は逆に根元から先端に向かって、各関節の位置を計算します（Fig. 6-44）。ある関節の回転角度をrad、根元側の隣接する関節からその関節に向かうベクトルを（px, py）とし、

c=cos (rad)
s=sin (rad)

とすると、先端側の隣接する関節に向かうベクトルは次のようになります。

(qx, qy) = (c*px-s*py, s*px+c*py)

したがって、関節の座標を（xi, yi）とすると、先端側の隣接する関節の座標は次のように計算できます。

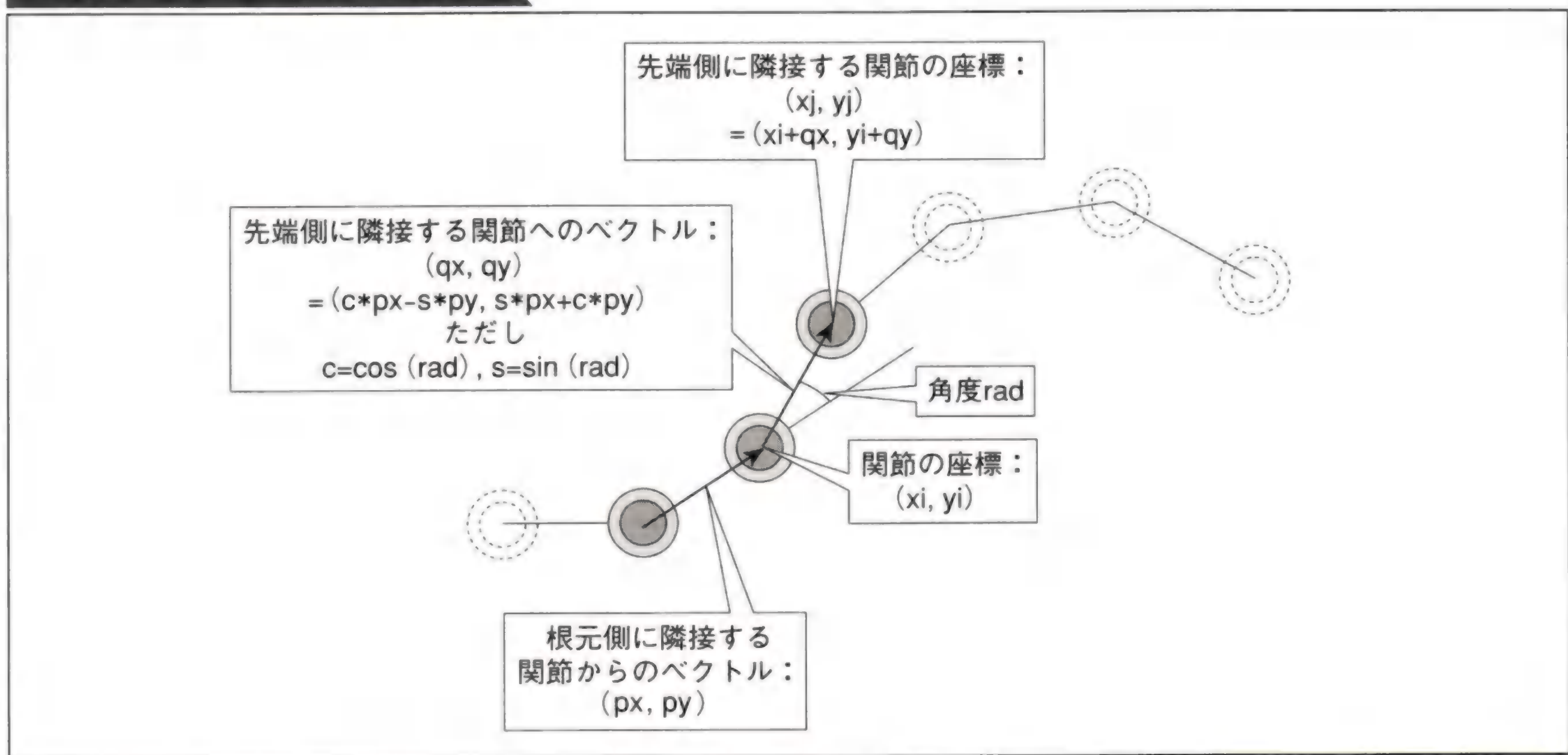
(xj, yj) = (xi+qx, yi+qy)

この方法を使って根元から先端に向かって順に関節の座標を求めれば、すべての関節の座標が求まります。全関節の座標が求まったら処理は完了です。あとは同じアルゴリズムで多関節を動かし続ければ、目標のあとを追いかけてグニャグニャと動く多関節を作ることができます。

長くなりましたが、多関節のアルゴリズムをプログラムにしたものがList 6-16です。前半部分がやや長いですが、全体としては意外にシンプルなプログラムにまとまります。

サンプル

多関節 → P. 321

Fig. 6-44 関節の位置の計算

List 6-16 多関節

```
#include <math.h>

void MoveJoints(
    int num_joint,          // 関節の数
    float x[], float y[],   // 関節の座標
    float rad[],            // 関節の回転角度
    float vrad,             // 関節が回転する速さ
    float lrad,             // 回転角度の限界値
    float dist,             // 関節間の距離
    float tx, float ty      // 目標の座標
) {
```

```
int i;
float c, s;

// 前半の処理：
// 先端から根元に向かって関節の角度を決める。
c=(float)cos(vrad);
s=(float)sin(vrad);
for (i=1; i<num_joint; i++) {
    float dxt, dyt;         // 関節から目標へのベクトル
    float dxn, dyn;         // 関節から先端へのベクトル
    float dxr=0, dyr=0;     // 右回りのベクトル
    float dxl=0, dyl=0;     // 左回りのベクトル
    float dpn, dpr, dpl;    // 内積(回らない、右回り、左回り)

    // 関節から目標へのベクトルの計算
    dxt=tx-x[i];
    dyt=ty-y[i];

    // 関節から先端へのベクトルと内積の計算
    // (回らない場合のベクトル)
    dxn=x[0]-x[i];
    dyn=y[0]-y[i];
    dpn=dxt*dxn+dyt*dyn;

    // 右回りのベクトルの計算：
    // 回転角度の限界を超えたときには回さない
    if (rad[i]+vrad<=lrad) {
        dxr=c*dxn-s*dyn;
        dyr=s*dxn+c*dyn;
        dpr=dxt*dxr+dyt*dyr;
    } else dpr=dpn;

    // 左回りのベクトルの計算：
    // 回転角度の限界を超えたときには回さない
    if (rad[i]-vrad>=-lrad) {
        dxl= c*dxn+s*dyn;
        dyl=-s*dxn+c*dyn;
        dpl=dxt*dxl+dyt*dyl;
    } else dpl=dpn;

    // 回転方向の選択：
    // 内積を比較して、回転を3通りのなかから選ぶ。
    // 先端を回転させて、新しい先端の位置を求める。
    if (dpr>dpn && dpr>dpl) {
        rad[i]+=vrad;
        x[0]=x[i]+dxr;
        y[0]=y[i]+dyr;
```

```
        }
        if (dpl>dpn && dpl>dpr) {
            rad[i]-=vrad;
            x[0]=x[i]+dxl;
            y[0]=y[i]+dyl;
        }
    }

    // 後半の処理：
    // 根元から先端に向かって関節の位置を求める。
    float px=dist, py=0, qx, qy;
    for (i=num_joint-2; i>=0; i--) {
        c=(float)cos(rad[i+1]);
        s=(float)sin(rad[i+1]);
        qx=c*px-s*py;
        qy=s*px+c*py;
        x[i]=x[i+1]+qx;
        y[i]=y[i+1]+qy;
        px=qx;
        py=qy;
    }
}
```

撃ち返し

破壊された瞬間に、自機に向かって敵が弾を撃ってくるものです（Fig. 6-45）。これは「撃ち返し弾」などと呼ばれます。

撃ち返しにはゲームの緊張感を増す働きがあります。また、ゲームの難易度を調整するための手段としても使われます。同じゲームでも2周目や3周目になると、1周目には何もしてこなかった敵が撃ち返し弾を撃つようになったりします。撃ち返し弾を使うと敵の攻撃パターンを変えなくても難易度を変えることができるので、こういった難易度調整に便利なのです。

撃ち返しを乱数と組み合わせるのも効果的です（Fig. 6-46）。必ず撃ち返しをするのではなく、ある確率でランダムに撃ち返すようにすると、回避方法がワンパターンではなくなるため、適度な緊張感が生まれます。撃ち返しの確率を高くしたり低くしたりすることで、難易度を変えることもできます。

List 6-17は撃ち返しに関するプログラムです。撃ち返し弾自体は「狙い撃ち弾」（→ P. 10）と同じ方法で作ることができます。

Fig. 6-45 撃ち返し

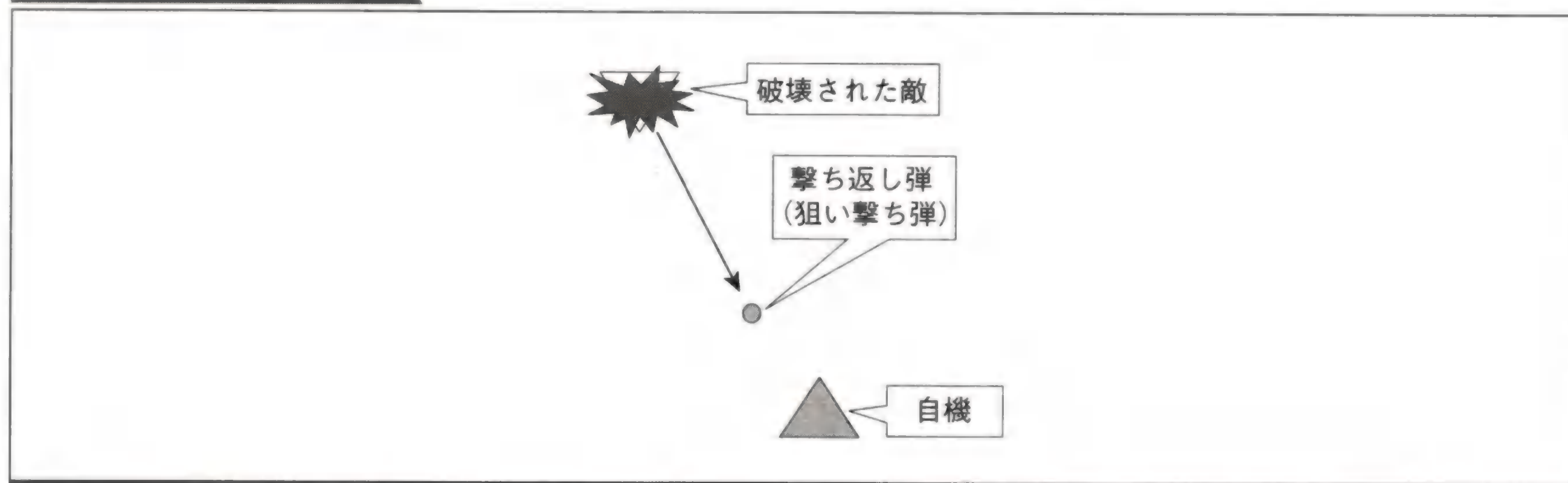
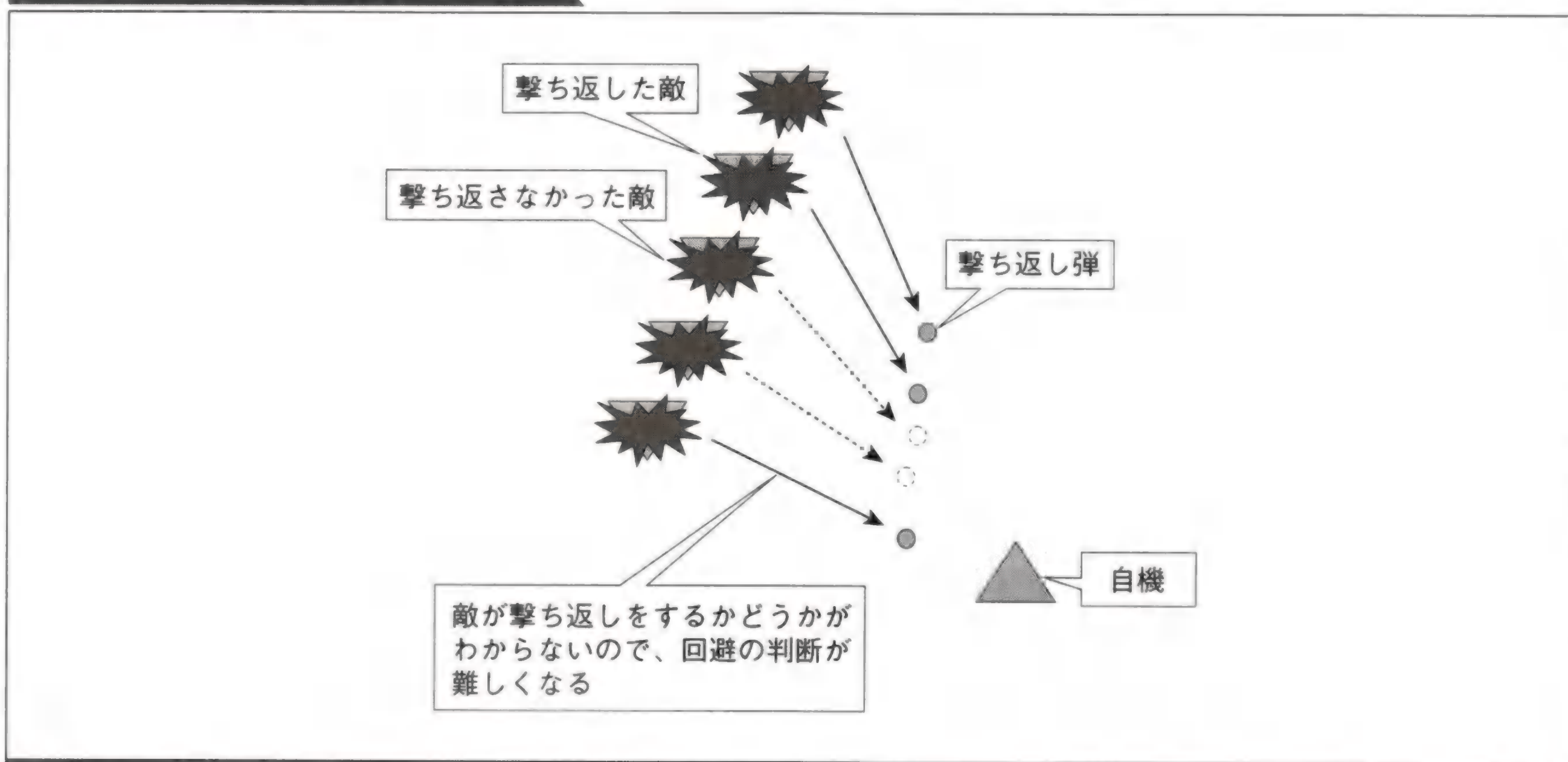



Fig. 6-46 乱数を使った撃ち返し

サンプル

撃ち返し →P. 320

List 6-17 撃ち返し

```
#include <stdlib.h>

void Revenge(
    float rate,           // 撃ち返しの確率
    float ex, float ey,   // 敵の座標
    float tx, float ty    // 目標(自機)の座標
) {
    // 敵が破壊されたかどうかの判定:
    // 具体的な処理はDestroyed関数で行うとする。
    if (Destroyed()) {

        // 撃ち返し:
        // 乱数を使って一定確率で撃ち返す。
        // 撃ち返し弾は「狙い撃ち弾」と同じ。
        // 発射の具体的な処理はAimingBullet関数で行うとする。
        if (rand()<=rate*RAND_MAX) {
            AimingBullet(ex, ey, tx, ty);
        }

        // 敵を消す:
        // 具体的な処理はDelete関数で行うとする。
        Delete();
    }
}
```

Stage 6 のまとめ ▶▶

敵の動きに関しては、弾を動かす処理の多くをそのまま適用することができます。上手にプログラムを書けば、同じ処理を使って楽にバリエーション豊かな敵を作ることができるでしょう。たとえば動きの処理を関数にまとめるとか、敵と弾に関する共通のベースクラスを作るなどといったプログラミング手法が有効です。

敵は弾に比べると絵を用意する手間がかかりますが、グラフィッカーにとっては敵のグラフィックこそ腕の見せどころでしょう。もちろん、プレイヤーの印象に強く残る敵やボスキャラを作るには、プログラマと連携して動きもよいものを作る必要があります。

というわけで、「敵はプログラマもグラフィッカーも腕の見せどころ!」というのが本章のまとめです。

Stage 07 ▶▶

背景 Scroll

まっ暗な宇宙を漂うだけのシューティングゲームも味わい深いものですが、やはり凝った背景や地形があったほうが、ゲームはずっと華やかになります。背景の表現方法はゲームによってさまざまですが、最近では3Dグラフィックを使って立体感のある背景を表示するゲームも増えてきました。

本章では主に背景や地形の表現方法とスクロールに関して解説します。スクロール方向は「上から下」（縦スクロール）と「右から左」（横スクロール）がポピュラーですが、8方向にスクロールしたり回転したりするゲームもあります。

横画面と縦画面

画面構成とスクロールは、シューティングゲームのゲーム性を決める重要な要素です。おおまかに分けて、画面構成には「横画面」と「縦画面」があります。それぞれの特徴は次のとおりです。

横画面
横方向：縦方向の比が「4：3」などの、横方向に長い画面です（Fig. 7-1）。横方向が遠くまで見えるので、横スクロールゲームに向いています。

縦画面
横方向：縦方向の比が「3：4」などの、縦方向に長い画面です（Fig. 7-2）。縦方向が遠くまで見えるので、縦スクロールゲームに向いています。

Fig. 7-1 横画面

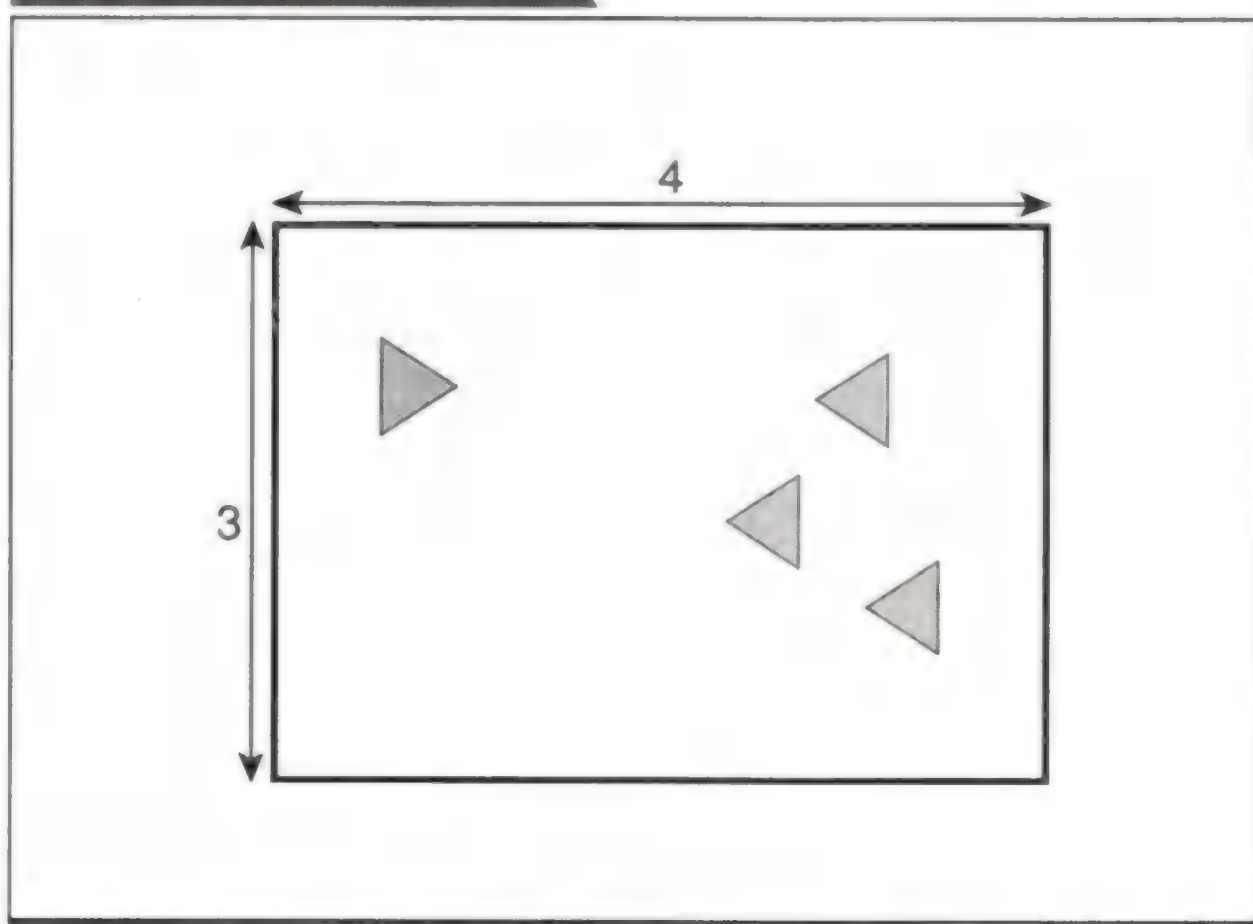
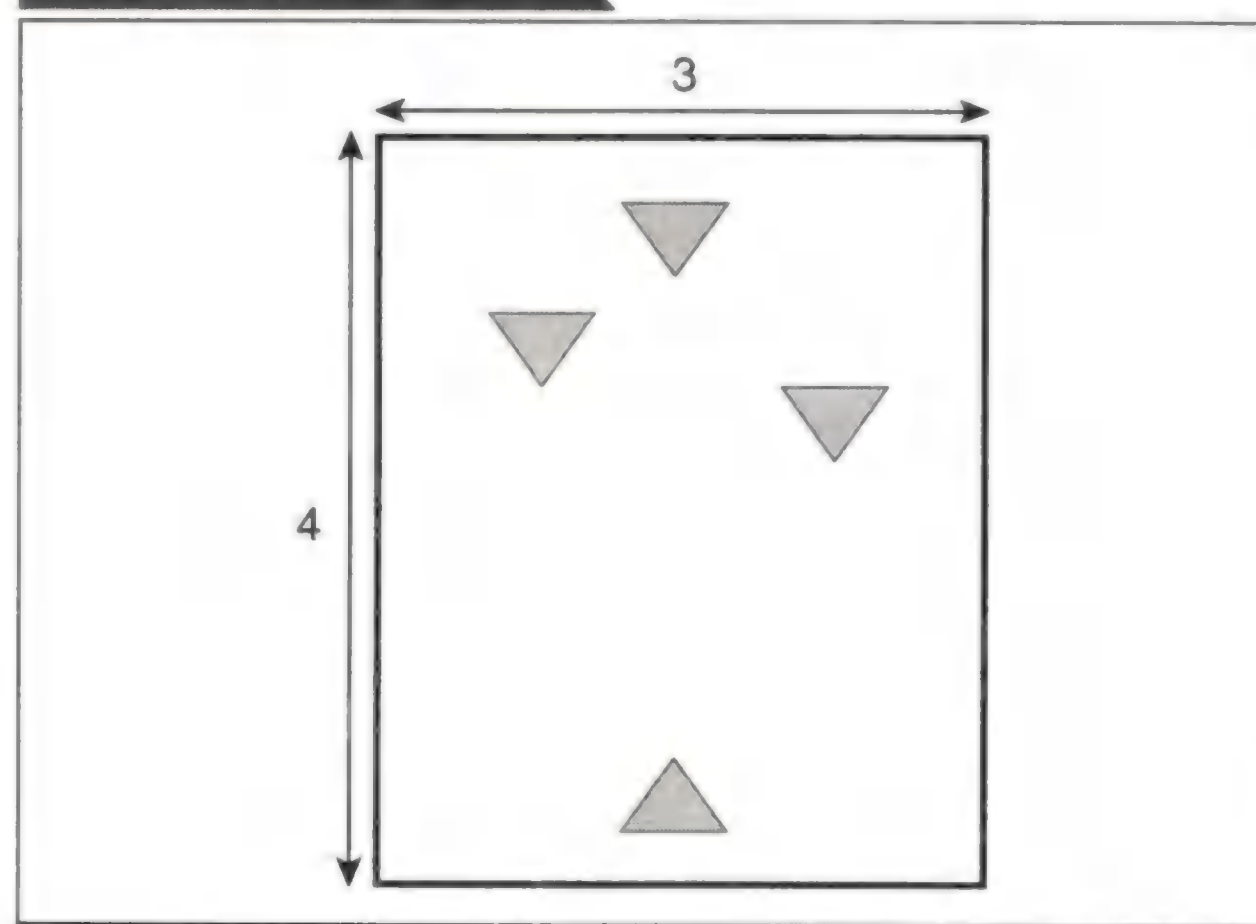



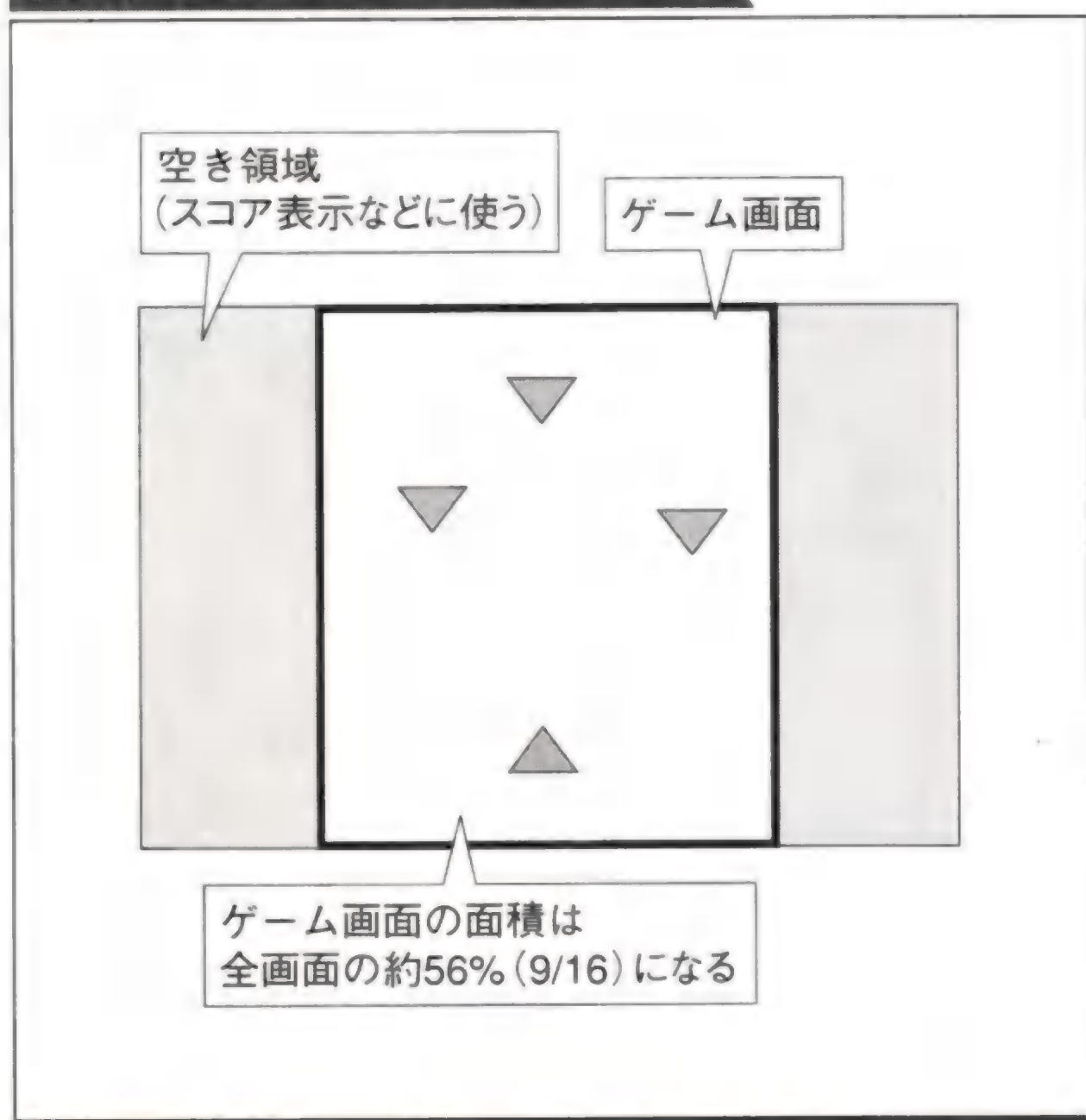
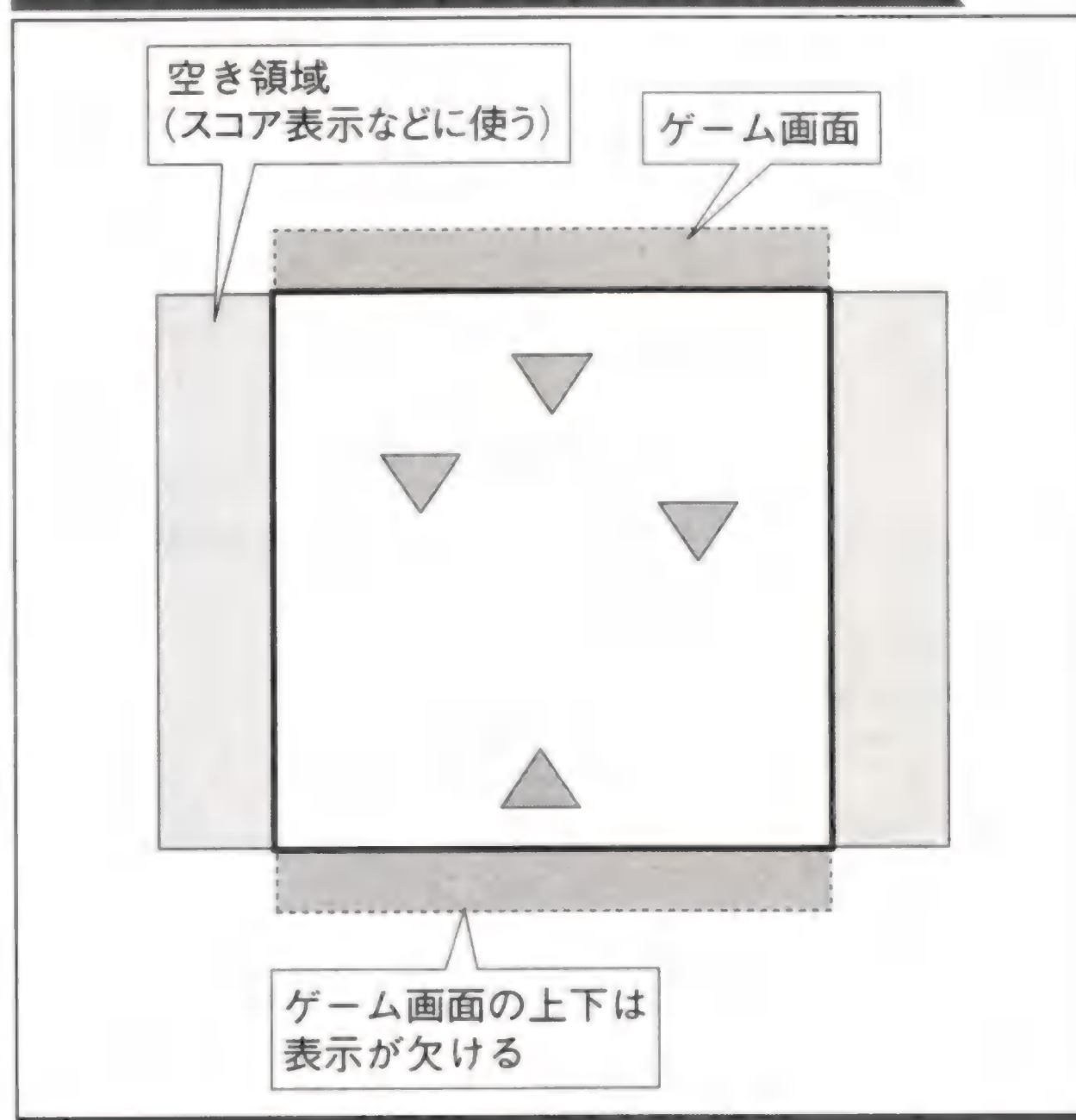
Fig. 7-2 縦画面

縦スクロールのアーケードゲームの多くは縦画面を使っていますが、これはアーケードゲーム筐体では画面を簡単に90度回転させることができるからです。家庭用ゲームの場合、家庭用TVは簡単には回転できないので、縦画面のゲームを移植する際には次のような工夫が必要になります。

横画面の一部を使う
横画面の一部（たとえば中央）を使って、縦画面を再現する方法です（Fig. 7-3）。この方法の欠点は画面が小さくなることです。たとえば、横と縦の比率が3：4の画面を4：3の画面の中央に表示するとなると、面積は約56%、つまり画面の約半分の大きさになってしまいます。TVの走査線の数にはかぎりがあるので、面積が小さくなると画像がつぶれてしまうのも問題です。

画面枠よりもやや大きめに表示することによって、画面の大きさを稼ぐこともあります(Fig. 7-4)。画面の上下端が少しずつ見えなくなってしまいますが、表示が欠けることを想定して画面端にはそれほど重要な情報を置かないことが普通なので、多少欠けていてもゲームはできます。また、もともとは画面の上下に表示されていたスコアや残機のような情報を、画面左右の空いた領域に追いやってしまうのも1つの方法です。

Fig. 7-3 横画面の一部を使う

空き領域
(スコア表示などに使う)
ゲーム画面
ゲーム画面の面積は
全画面の約56%(9/16)になる

Fig. 7-4 画面枠よりも大きめに表示する

空き領域
(スコア表示などに使う)
ゲーム画面
ゲーム画面の上下は
表示が欠ける

ゲームを横画面にアレンジする

ゲーム自体を横画面にアレンジしてしまう方法です(Fig. 7-5)。これは移植方法としては手間がかかるわりに、原作とはゲーム性が大きく変わってしまうのでプレイヤーの評判も悪いという、あまり利点がない方法でもあります。

仮想的に縦画面を表現するために、プレイヤーが縦方向に画面を動かせるようにすることもあります(Fig. 7-6)。自機を上に動かすと表示も上に動き、下に動かすと表示も下に動きます。雰囲気としては縦画面に近くなりますが、やはり遠くが見えないのは致命的で、ゲーム性は元のゲームとは大きく違ってしまいます。

縦画面をそのまま横画面に表示する

元の縦画面に手を加えずに、そのまま横画面に表示する方法です(Fig. 7-7)。これは元の画面を完全に再現することができますが、家庭用TVは横置きが普通なので、表示が90度回転されてしまうのが最大の問題点です。しかしこの方法は見た目もゲーム性も完全に移植することが可能なので、マニアを中心に支持されています。

寝そべって遊べば90度回転した画面でもなんとか遊べるのですが、慣れないとどうにも落ち着きません。いちばんよいのは、回転機能（ピボット機能）を備えた液晶ディスプレイなどに画面を出力して、ディスプレイを90度回転させて遊ぶことです。ゲームによってはコントローラの入力方向を90度回転させて、横画面・横スクロールのゲームとして遊べるモードをつけていることもあります。

Fig. 7-5 横画面へのアレンジ

全体に横に伸びたような
ゲーム画面になる

Fig. 7-6 縦方向に画面を動かす

表示されない範囲
（全体としては縦画面と同じ比率）

表示される範囲
（横画面）

自機の動きに応じて
表示範囲が上下に動く

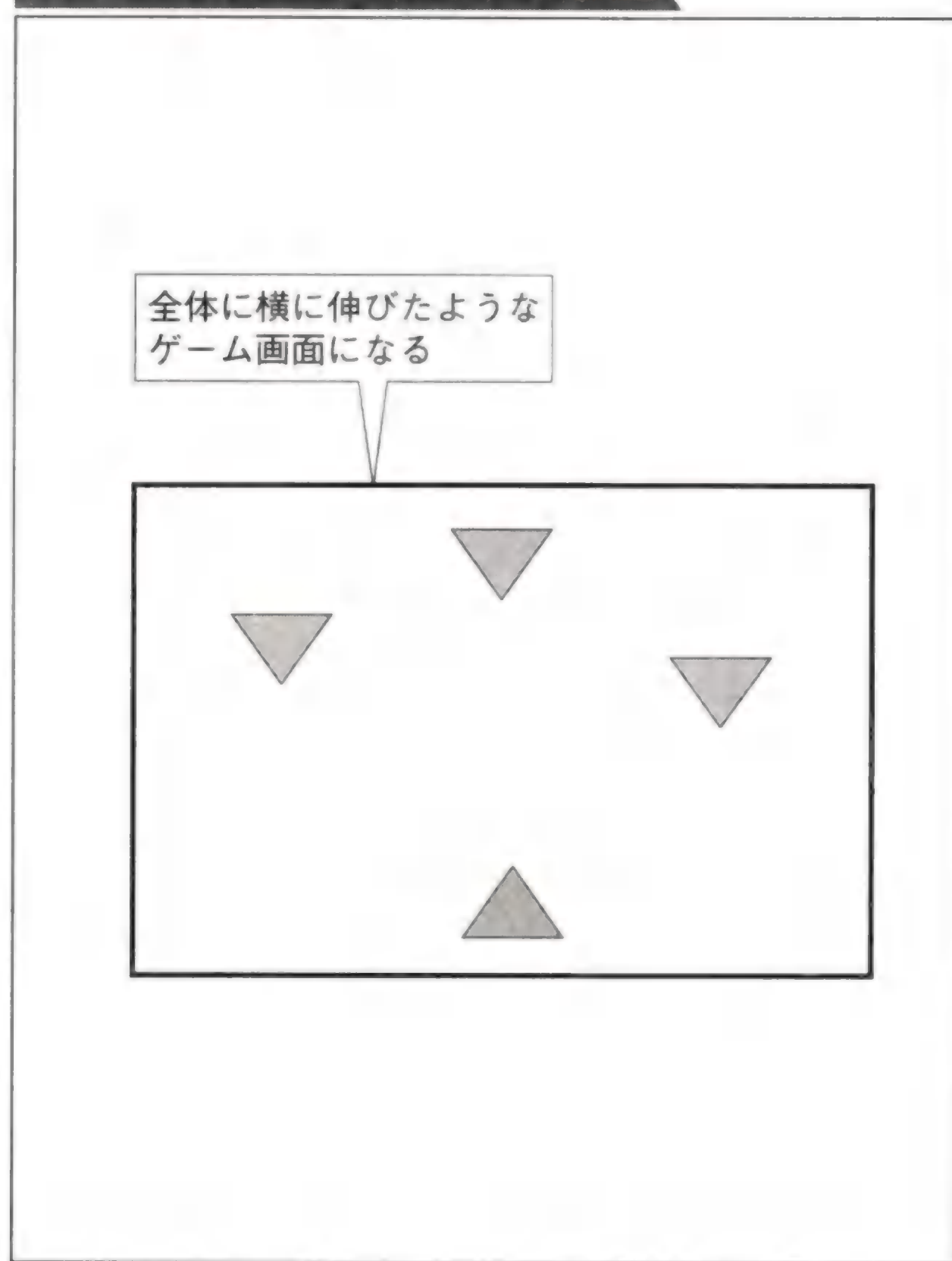
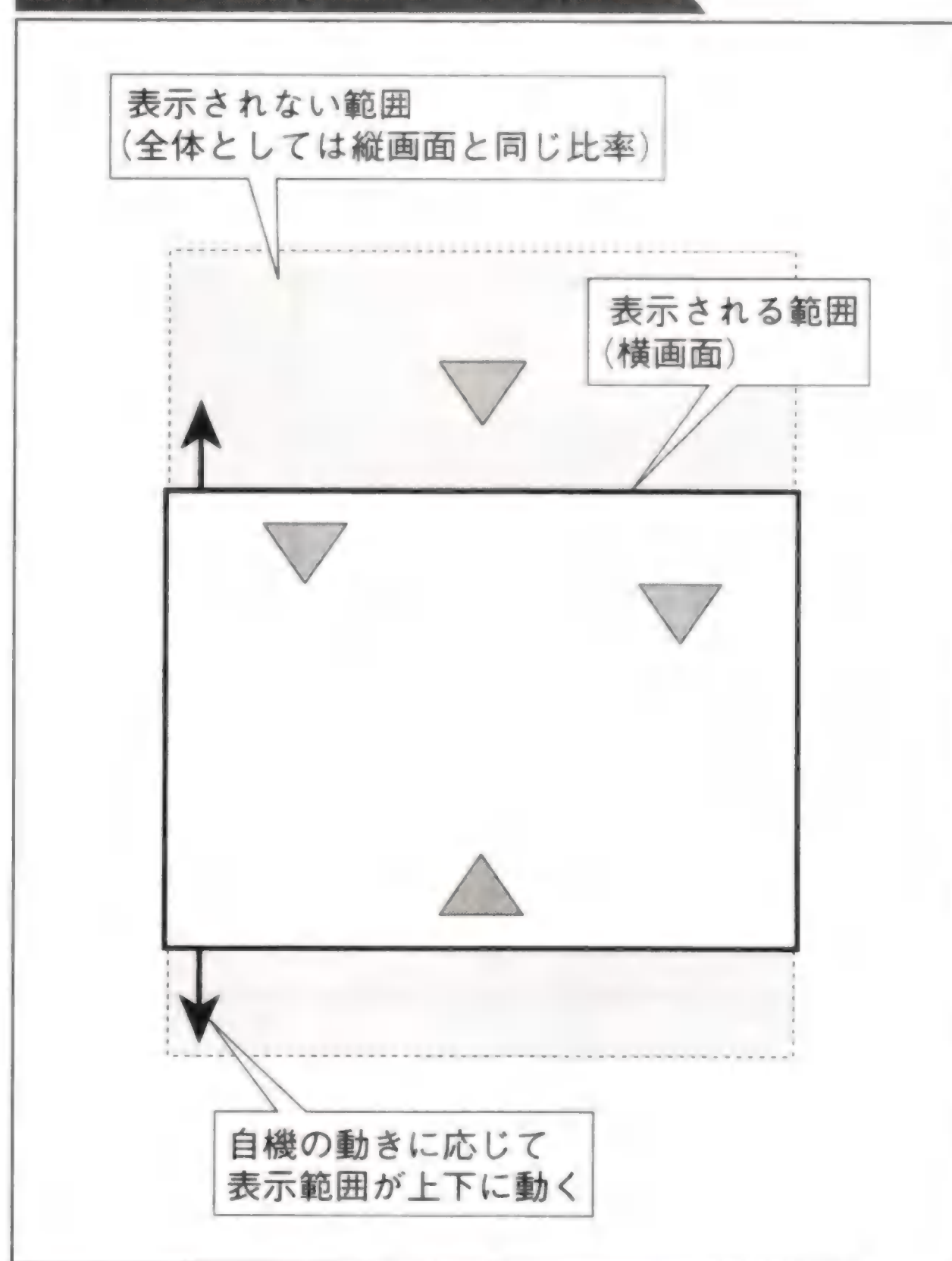
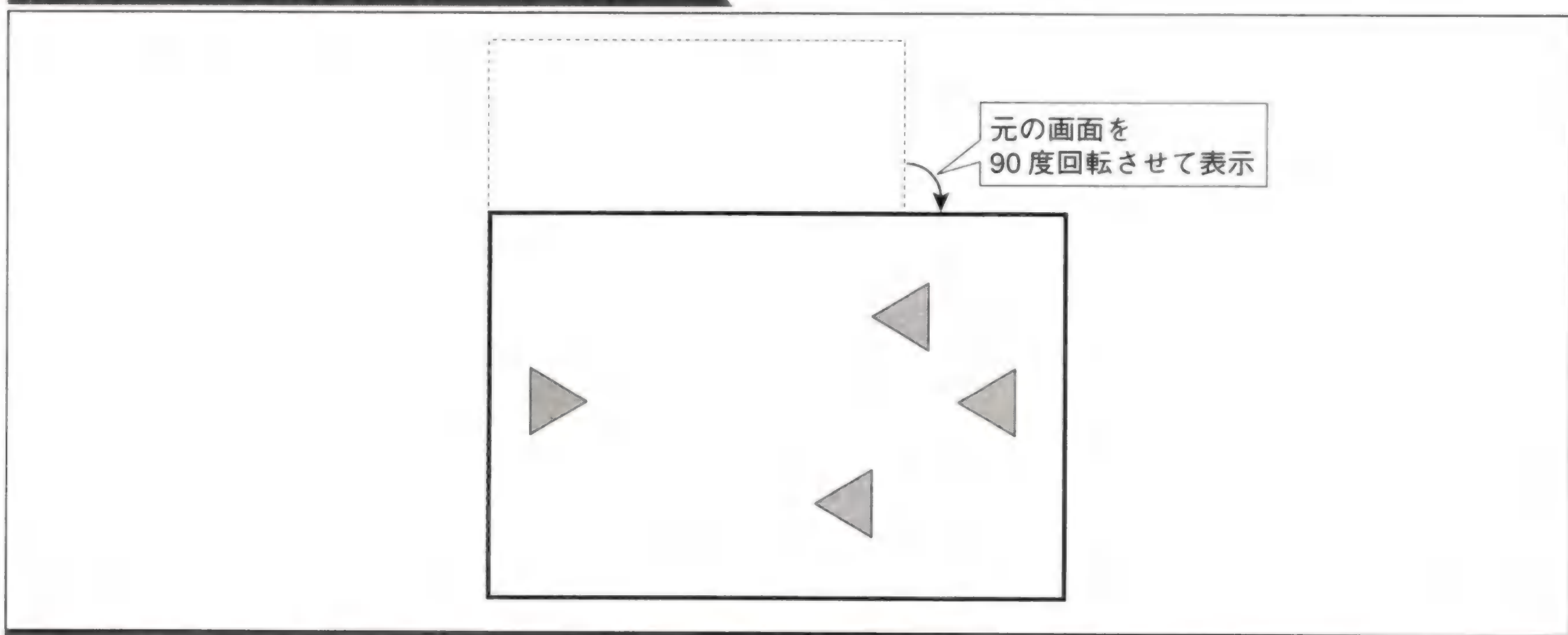



Fig. 7-7 縦画面をそのまま横画面に表示する

元の画面を
90度回転させて表示

液晶ディスプレイへ簡単に画面が出力できるゲーム機は少ないので、その点ではVGA出力を備えた「ドリームキャスト」はシューティングゲーマー御用達のゲーム機だといえます。今でも（2004年4月現在）ドリームキャストに最新のシューティングゲームが移植され続けているのは、アーケード版シューティングゲームの多くがNAOMI基板（ドリームキャストと互換性があるアーケード基板）で作られていることもありますが、縦画面ゲームをそのまま移植しても快適に遊べるような仕様のマシンだからでもあるでしょう。ドリームキャストが生産中止になったのは、シューターにとっては大きな損失だと思います。

※

このように、どの方法をとっても縦画面ゲームを横画面にアレンジするのはたいへんです。新しく縦スクロールゲームを作るときには、横画面にするのか、縦画面にするのか、あるいは横画面の一部を使って縦画面を表現するのかを、念入りに検討する必要があります。その点、PCでシューティングゲームを作る場合には、横画面の一部を使って縦画面ゲームを作っても十分な解像度を得られるのがよいところです。

縦スクロールゲームを最初から横画面用に作るのも1つの方法です。「ソニックウィングス（→ P. 329）」シリーズ（2以降）や「ギガウィング（→ P. 325）」シリーズなどは縦スクロールゲームですが、もともと横画面用に作られています。

スクロール

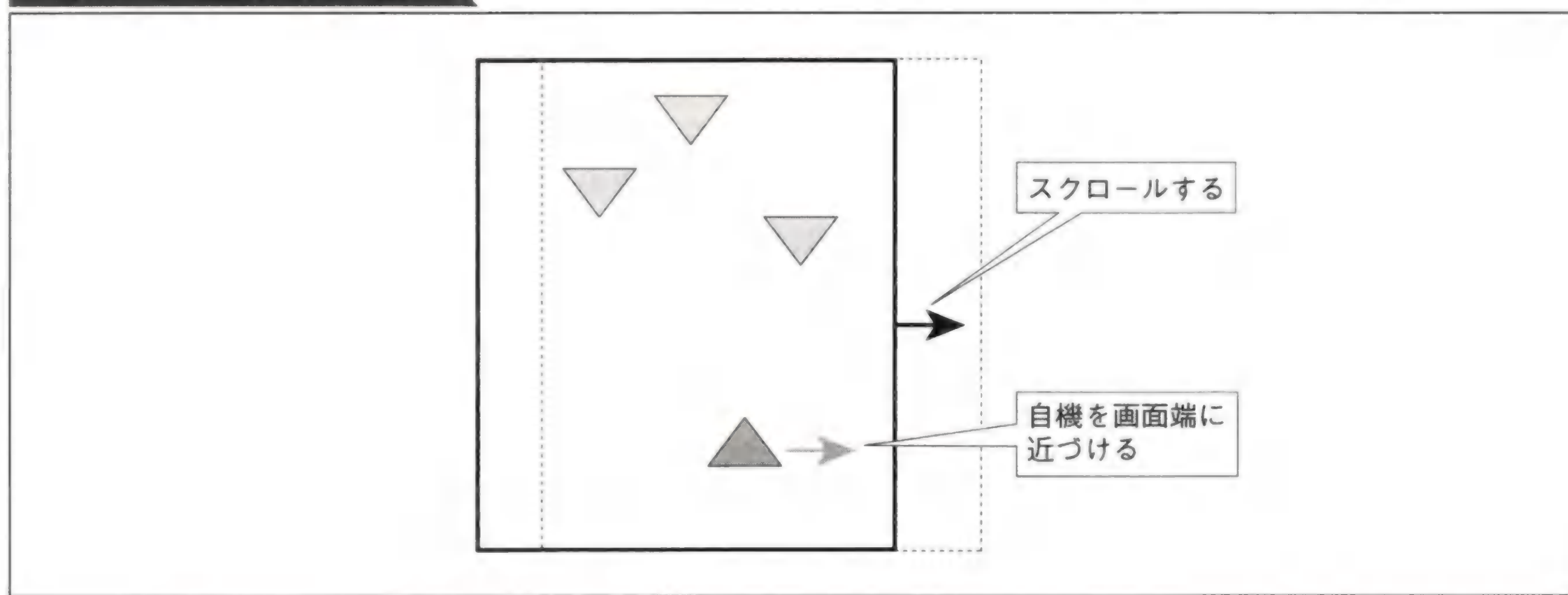
背景や地形を流しながら表示することによって、マップ上を進んでいるような雰囲気を演出するのがスクロールです。ちなみに「スクロール（scroll）」という言葉は魔法使いの呪文などが書いてある「巻き物」を意味します。長いマップを少しずつスクロール表示するのは、確かに巻き物を読む様子に似ています。

スクロールには「強制スクロール」と「任意スクロール」という2つの方式があります。強制スクロールというのは、プレイヤーの操作に関係なく、自動的に背景がスクロールしていくものです。任意スクロールでは、プレイヤーがスクロールを制御することができます。多くの任意スクロールゲームでは、自機を画面端に近づければ、画面をその方向にスクロールさせることができます（Fig. 7-8）。

スクロール方向にはさまざまなタイプがあります。ここでは代表的なスクロール方向と、その特徴について解説します。

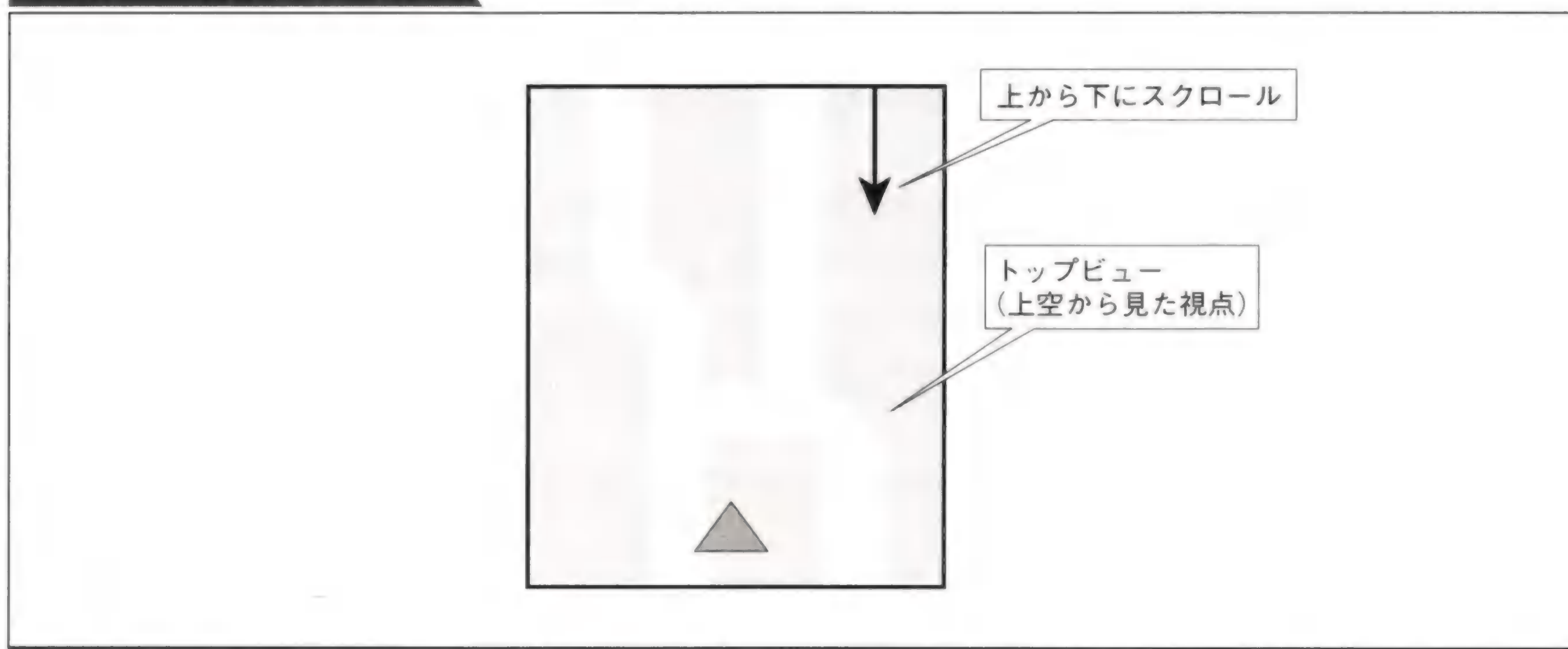
固定画面

スクロールしないものです。「ギャラガ（→ P. 325）」などのように、基本的には固定画面で、星などの簡単な背景だけをスクロールさせていることもあります。固定画面ゲームとしては「スペースインベーダー（→ P. 328）」や「ギャラクシアン（→ P. 326）」などが有名です。「プーヤン（→ P. 333）」なども固定画面ゲームに分類されるでしょう。

Fig. 7-8 任意スクロール

縦スクロール

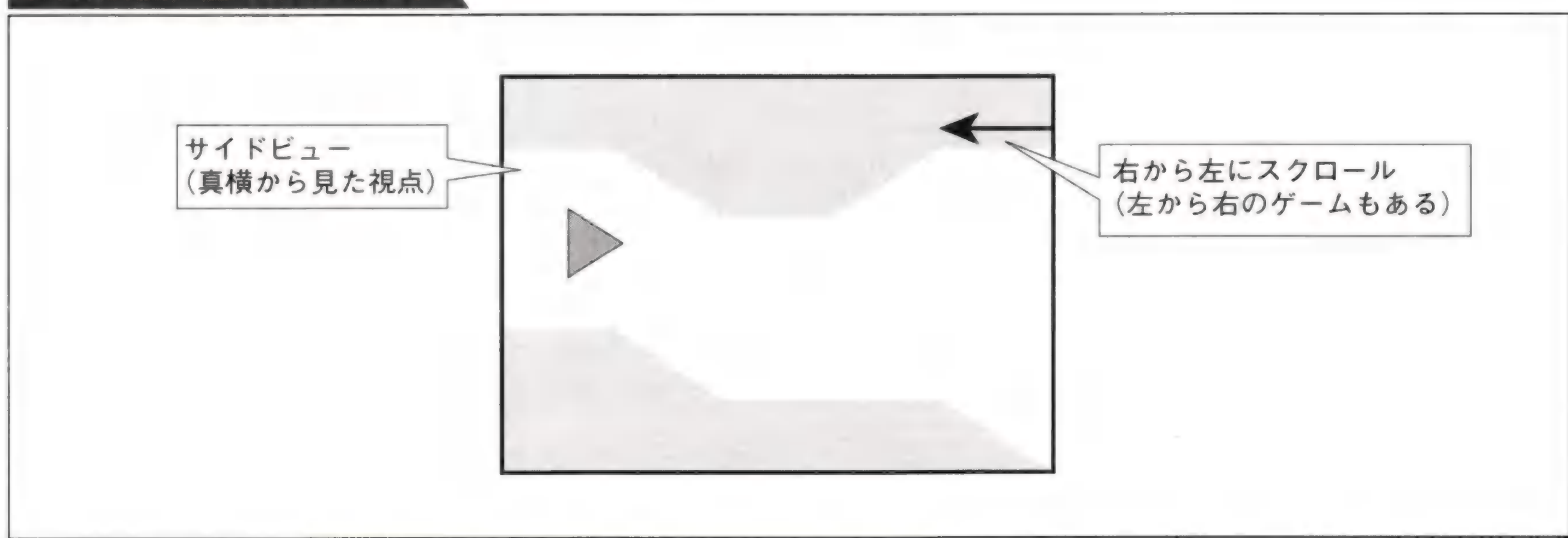
上下方向へのスクロールです（Fig. 7-9）。多くの縦スクロールゲームはトップビュー（上空から見た視点）になっています。縦スクロールゲームは多数ありますが、「ゼビウス（→ P. 329）」などが有名です。

ほとんどの縦スクロールゲームは上から下にスクロールします。これは画面を見たときに、自機の正面とプレイヤーの正面が一致するからでしょう。下から上にスクロールするゲームは少ないですが、横スクロールゲームで縦穴を下っていくようなステージには下から上への強制スクロールが見られます。

Fig. 7-9 縦のスクロール

横スクロール

左右方向へのスクロールです（Fig. 7-10）。多くの横スクロールゲームはサイドビュー（真横から見た視点）を採用しています。縦スクロールゲームと同様に、横スクロールゲームも多数ありますが、「グラディウス（→ P. 326）」や「R-TYPE（→ P. 323）」などが有名どころです。

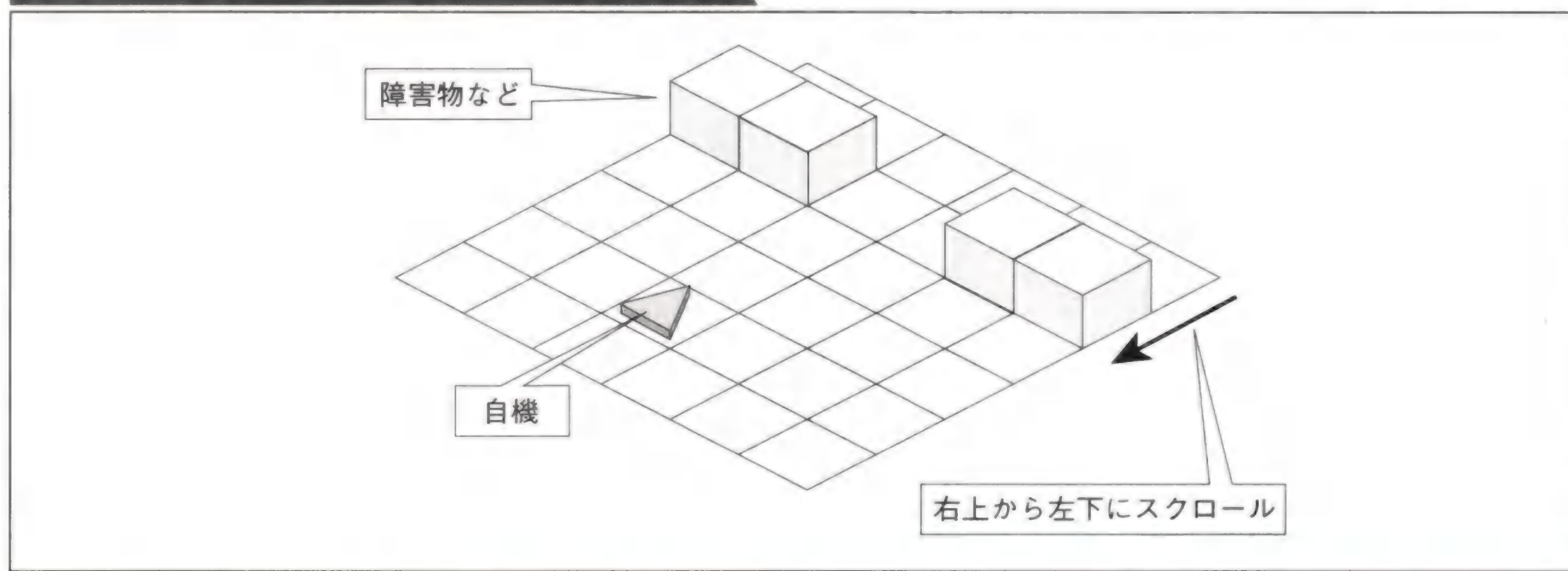
ほとんどの横スクロールゲームは右から左にスクロールします。「スカイキッド（→ P. 328）」のように左から右にスクロールするゲームもありますが、少数派です。右→左のスクロールが好まれる正確な理由はわかりませんが、アーケードゲーム筐体の場合には左側に1P（1人目のプレイヤー）が座り、右→左スクロールでは自機が左側にいることが多いので、相性がよいのかもしれません。

Fig. 7-10 横スクロール

斜めスクロール

画面全体を斜めから見下ろしたような視点（クォータービュー）で、斜めにスクロールするものです（Fig. 7-11）。多くの場合は右上から左下にスクロールします。現在ではシューティングゲームにも3Dグラフィックを使うのが当たり前ですが、3Dグラフィックが一般的でなかった時代には、クォータービューは「3Dっぽい画面」を表現するための代表的な技法でした。

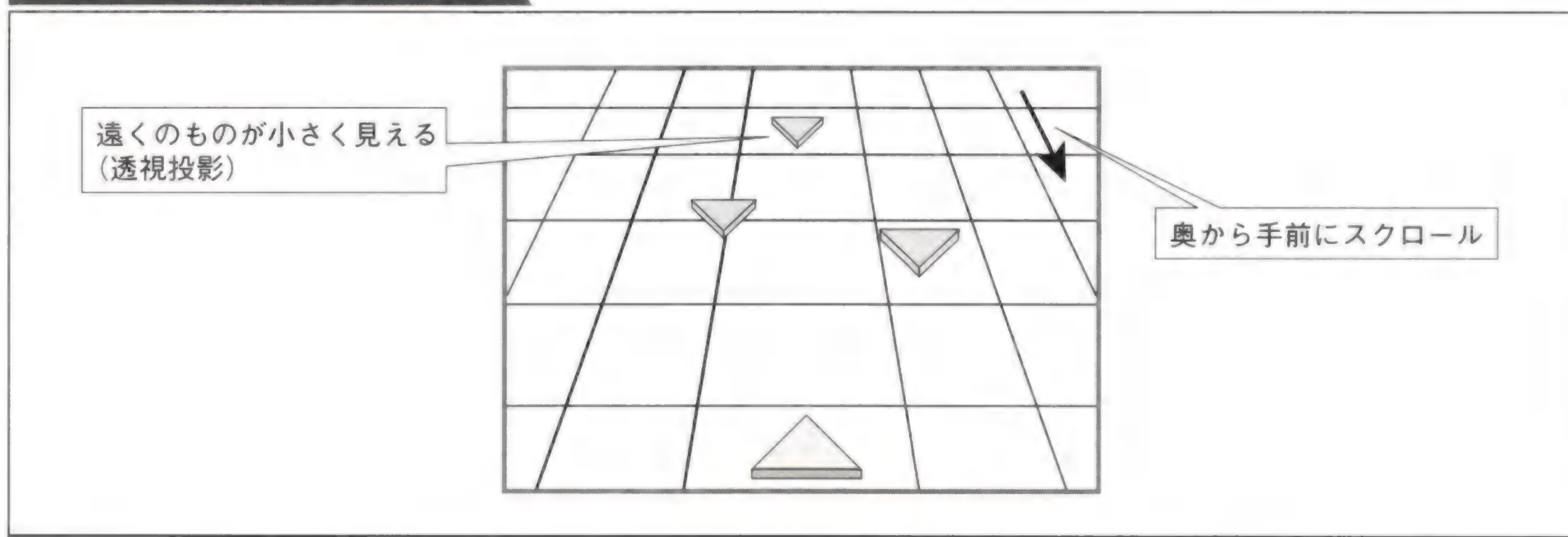
クォータービューの表示は3D的ですが、自機の動きは2D的です。「ザクソン（→ P. 327）」の場合には、自機の奥行きは固定で上下左右に動きます。「ビューポイント（→ P. 332）」の場合には、自機の高度は固定で前後左右に動きます。

クォータービューのよいところは、立体感のある画面が作れることと、縦スクロールに比べて横画面でも前方の見通しがよいことです。欠点は、グラフィックを描くのに手間がかかることと、操作が独特なのでオーソドックスな縦スクロールゲームや横スクロールゲームに比べるとプレイヤーの好みが分かれやすいことです。

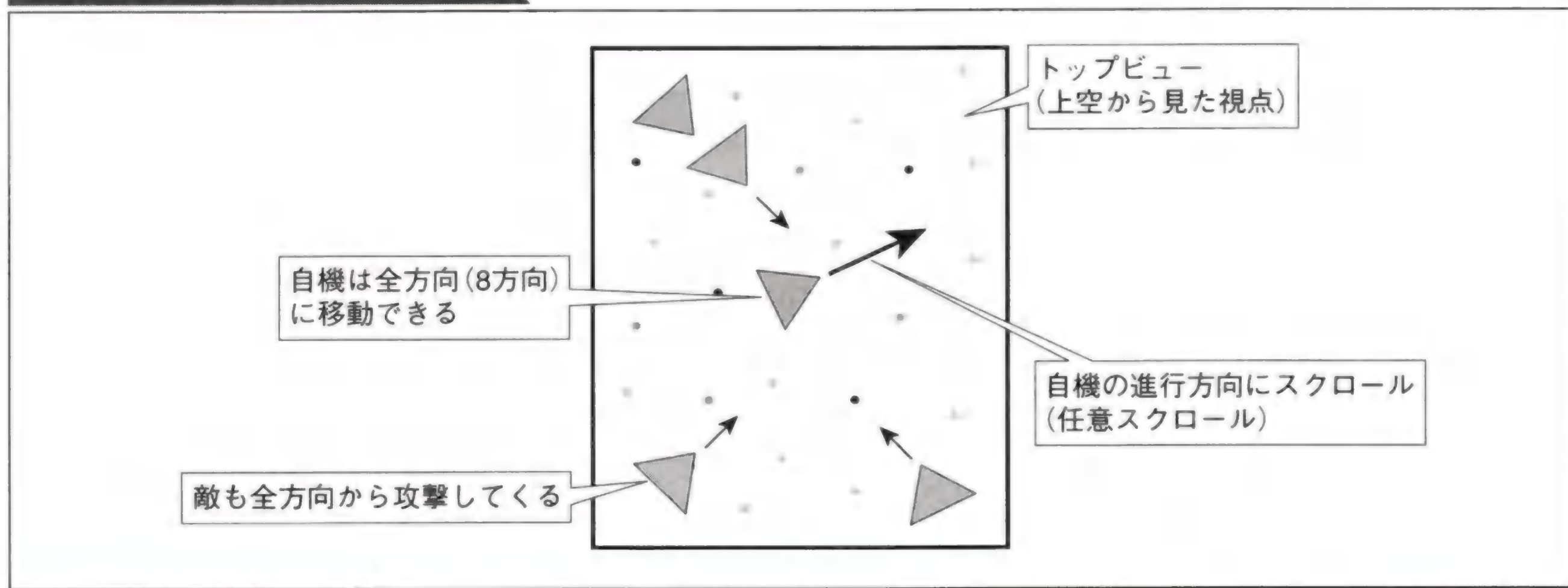
Fig. 7-11 斜めスクロール(クォータービュー)

奥行きスクロール

画面全体を斜めから見下ろしたような視点で、奥に向かってスクロールするものです(Fig. 7-12)。クォータービューでは遠くのものが小さく見えないのですが(平行投影)、奥行きスクロールの場合には遠くのものが小さく見えるのが違います(透視投影)。

奥行きスクロールゲームには「レイストーム(→ P. 335)」「レイクライシス(→ P. 334)」「ゼビウス3D/G(→ P. 329)」などがあります。奥行きスクロールの利点は、縦スクロールふうのゲームを横画面で作ることができ、しかも前方の見通しがよいことです。ただし、2Dグラフィックだけで奥行きスクロールを表現するのは難しいので、3Dグラフィックを使う必要があります。

Fig. 7-12 奥行きスクロール

全方向スクロール

全方向にスクロールするものです（Fig. 7-13）。「トップビュー＋全方向への任意スクロール」という組み合わせがよく見られます。全方向スクロールゲームの有名どころは「タイムパイロット（→ P. 329）」や「ボスコニアン（→ P. 334）」です。「タイムパイロット」はトップビューとサイドビューが微妙に混じったような視点を採用しています。また、戦車系のゲームでも全方向スクロールを使うことがあります。

Fig. 7-13 全方向スクロール

このほか、疑似3D（2Dグラフィックで3Dグラフィックふうの表現をする手法）を使って、自機の背後から見た視点で画面奥に向かって直進するタイプのゲームもあります。「スペースハリアー（→ P. 328）」や「アフターバーナー（→ P. 324）」などがこの例です。こういったゲームは、一般的な縦スクロールゲームや横スクロールゲームとはかなりゲーム性が違うので、プレイヤーの好みは分かれますが、本格的な3Dグラフィックが普及する少し前には流行していました。

普通の縦スクロールゲームや横スクロールゲームでも、最近では3Dグラフィックを使ったものが多くなりました。こういったゲームの多くは、絵は3Dグラフィックで描いているものの、ゲーム性は2Dゲームと変わりません。3Dグラフィックを使うと、キャラクターを回転させたり、背景に立体感を出したりするのが簡単なので、個人がシューティングゲームを作る場合にも3Dグラフィックを使うのは有効な方法です。

背景の表示

スクロールさせる背景や地形は、矩形の小さな部品を敷き詰めて作ることがほとんどです（Fig. 7-14）。個々の部品は「チップ」と呼ばれることがあります。

チップを並べて作った背景は、チップ番号の配列としてデータ化することができます（Fig. 7-15）。たとえば、1ステージの背景が20×400のチップで構成されているときには、20×400の2次元配列を用意して、チップ番号を記録しておきます。1次元配列を使ってもかまいません。

Fig. 7-14 チップを使った背景の作成方法

背景の部品
（チップ）

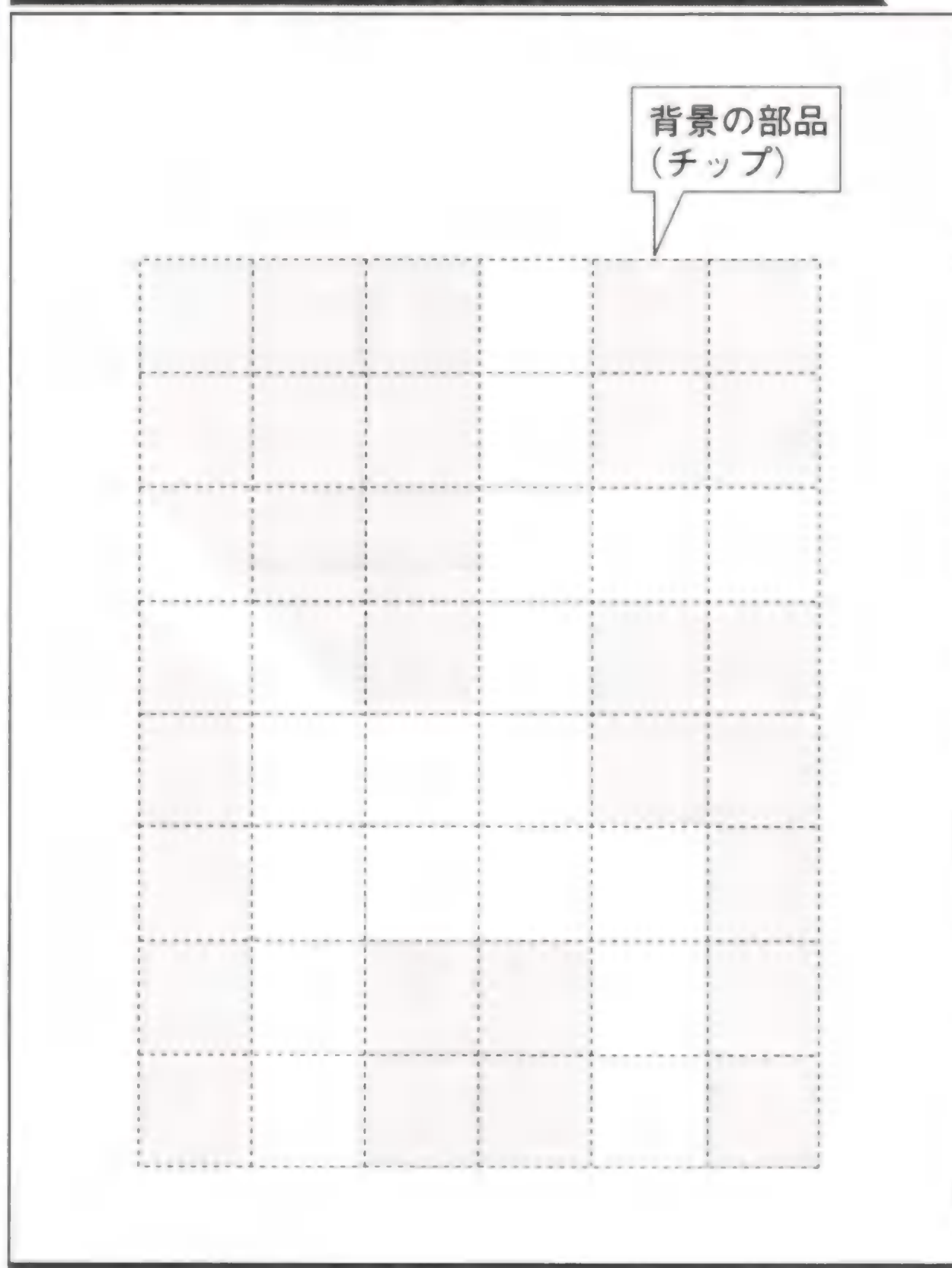



Fig. 7-15 チップを使った背景のデータ化

チップ番号1
チップ番号0
チップ番号2
チップ番号3

1	1	1	0	1	1
1	1	1	0	1	1
2	1	1	0	0	0
3	2	1	0	1	1
1	0	1	0	1	1
1	0	0	0	2	1
1	0	1	1	0	1
1	0	1	1	0	1

チップ番号の配列としてデータ化する：
map [YMAX] [XMAX]
= {{1, 1, 1, 0, 1, 1}, ..., {1, 0, 1, 1, 0, 1}}
（YMAX、XMAXはY方向、X方向のチップ数）

背景を表示するときには、スクロールで表示中の位置を考慮して、画面内に入るチップだけを描画します（Fig. 7-16）。画面左上に対応する背景上の位置を（sx, sy）、画面の大きさを（sw, sh）、個々のチップの大きさを（cw, ch）とすると、画面に入るチップの範囲（配列のインデックス）は、

(sx/cw, sy/ch)

から、

((sx+sw-1) /cw, (sy+sh-1) /ch)

までの矩形領域になります。ここではすべての変数は整数型として、割り算の際に端数を切り捨てることにします。

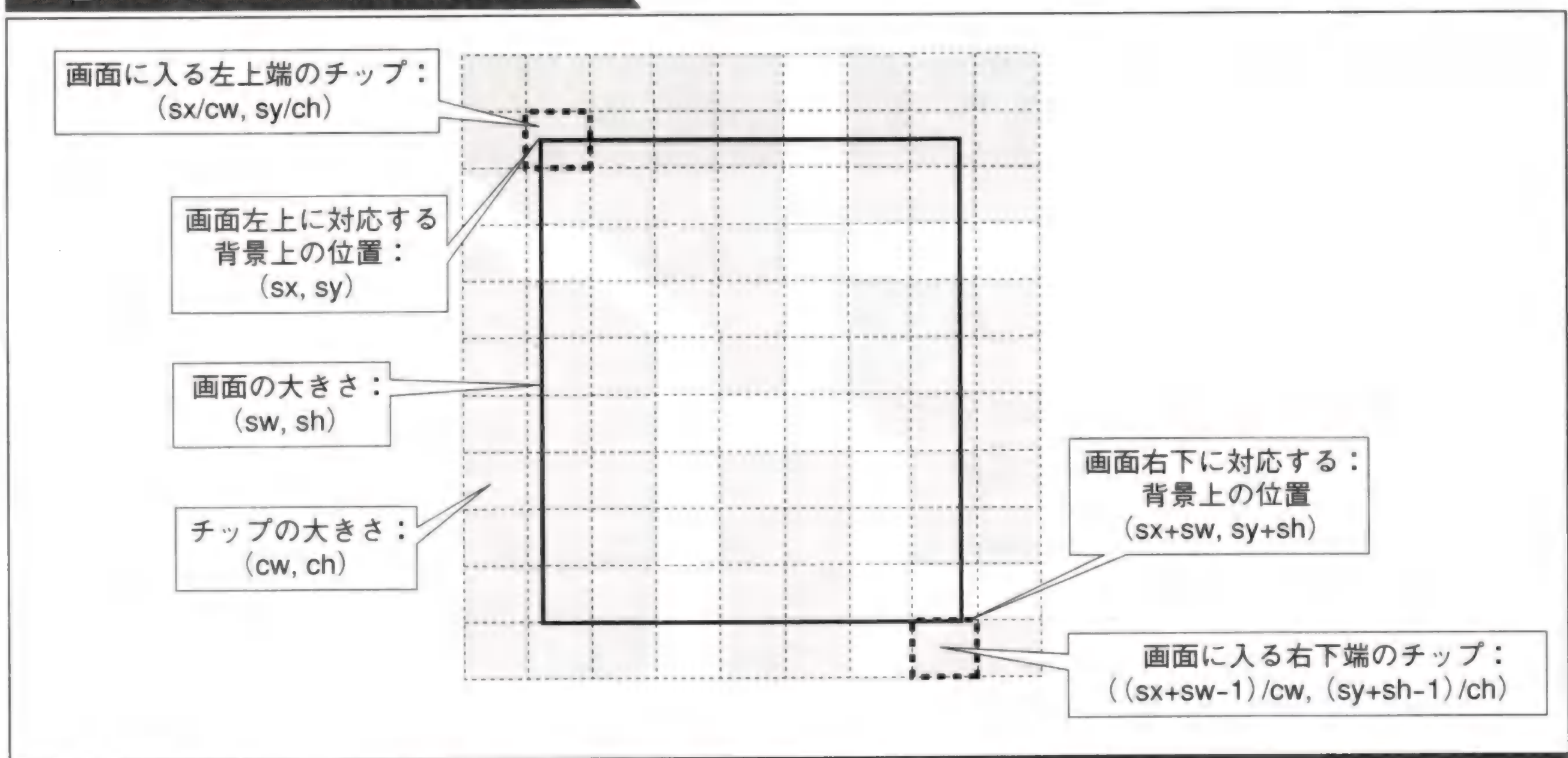
また、データ上の位置が[i][j]のチップを表示する画面上の位置は次のようになります。

(j*cw-sx, i*ch-sy)

List 7-1はチップを使った背景を表示するプログラムです。画面の位置から描画する範囲を求め、範囲内のチップを順に描画します。

サンプル

背景の表示 → P. 321

Fig. 7-16 チップを使った背景の表示

List 7-1 チップを使った背景の表示

```
// チップ数(X方向、Y方向)
#define XMAX 20
#define YMAX 400

// 背景の表示
void DrawBG(
    int sx, int sy,        // 画面左上に対応する背景上の位置
    int sw, int sh,        // 画面の大きさ
    int cw, int ch,        // チップの大きさ
    int map[YMAX][XMAX]    // 背景データ(チップ番号の配列)
) {
```

```
    // 描画するチップの範囲
    int x0=sx/cw, y0=sy/ch;                   // 左上端のチップ
    int x1=(sx+sw-1)/cw, y1=(sy+sh-1)/ch;     // 右下端のチップ

    // チップの描画：
    // 各チップの描画はDrawChip関数で行うとする。
    for (int i=y0; i<=y1; i++) {
        for (int j=x0; j<=x1; j++) {
            DrawChip(map[i][j], j*cw-sx, i*ch-sy);
        }
    }
}
```

多重スクロール

異なる速度でスクロールする背景を何枚か重ねて、画面の奥行きを表現する技法です。たとえば、次のような具合に使います(Fig. 7-17)。

- **手前に当たり判定のある地形を表示**
- **奥に当たり判定のない星空を表示**

手前の背景を速く、奥の背景を遅くスクロールさせれば、画面に奥行き感が出ます。

昔は多重スクロールを行うために、背景のスクロールと重ね合わせを行う専用のハードウェアを使いましたが、現在では背景を単に重ねて描くことによって多重スクロールを表現するほうが多いでしょう。多重スクロールを表示するには、奥の画面を先に描き、手前の画面をあとで描きます(Fig. 7-18)。ただし、ハードウェアで3Dグラフィックを描画し、かつZバッファを使う場合には、逆の順番で描いたほうが効率的な場合もあります。

多重スクロールの処理はList 7-2のようになります。スクロールする各背景の描画は「背景の表示」(→ P. 270)で解説した方法と同じです。

Fig. 7-17 多重スクロール

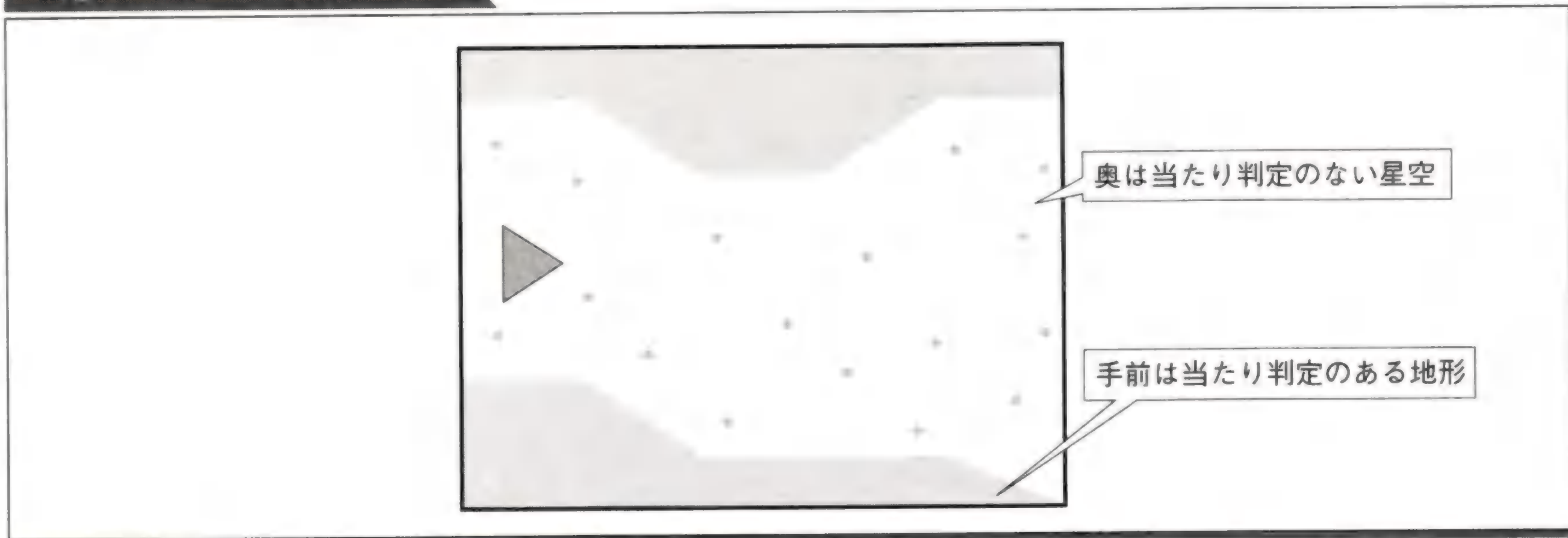
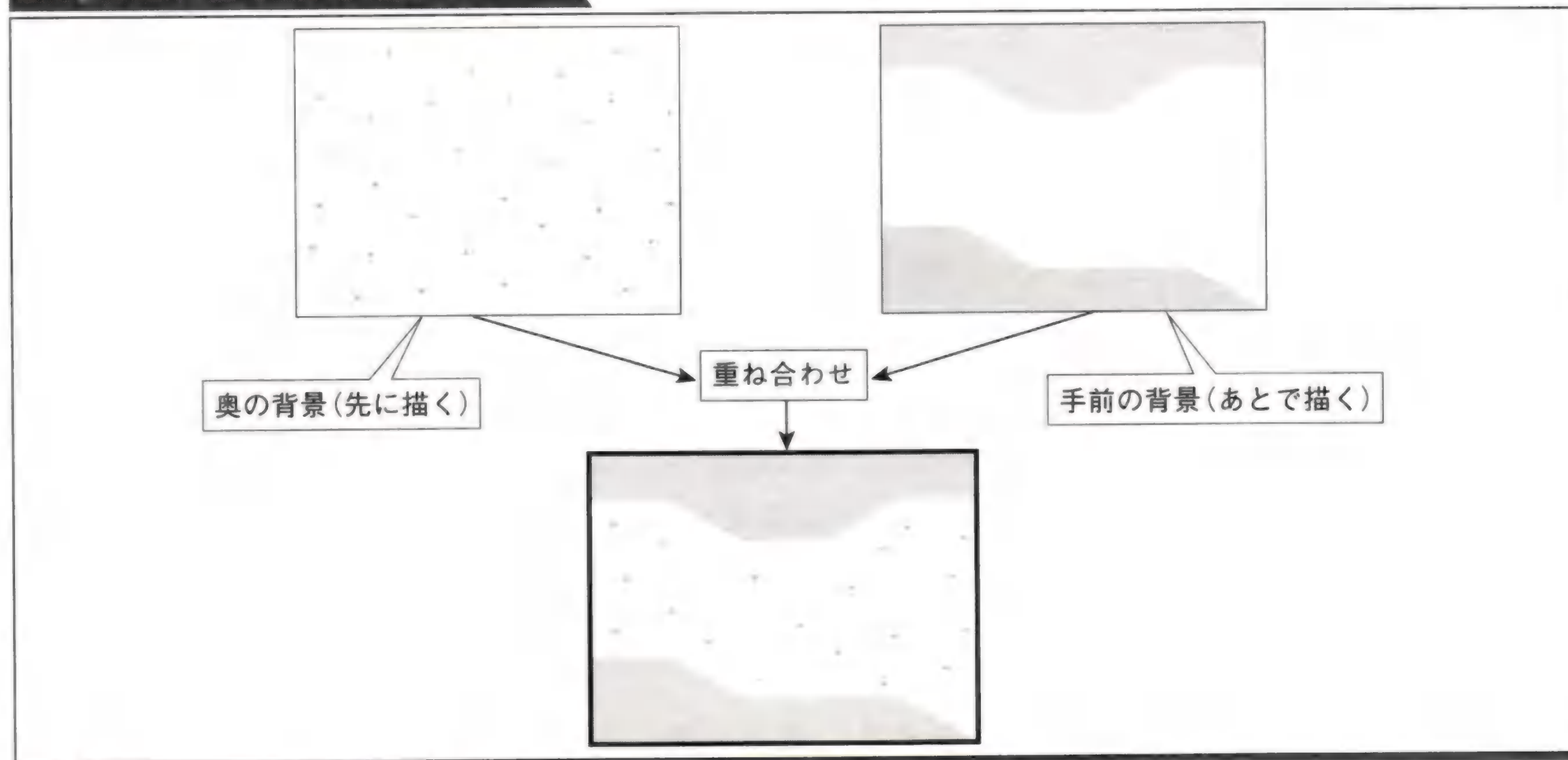



Fig. 7-18 多重スクロールの描画

サンプル

多重スクロール → P. 321

List 7-2 多重スクロール

```
void OverlaidScroll(
    int num_bg,             // 背景の数
    int sx[], int sy[],     // 各背景上の表示位置
    int svx[], int svy[]    // 各背景のスクロール速度
) {
    // 全背景の描画：
    // 各背景の表示位置を順に更新したあとに表示する。
    // 表示の具体的な処理はDrawBG関数で行うとする。
    for (int i=0; i<num_bg; i++) {
        sx[i]+=svx[i];
        sy[i]+=svy[i];
        DrawBG(i);
    }
}
```

星のスクロール

ピクセル（点）などで描かれた小さな星をスクロールさせる表現方法です（Fig. 7-19）。チップを使った背景に比べて簡単に表示することができます。また、星の速度を何通りかに変えれば、「多重スクロール」（→ P. 272）と似たような効果で画面に奥行きを出すことができます。

星をスクロールさせるには、表示する星の数だけ座標、速度、色を記録しておき、これらの情報に基づいて星を描きます（Fig. 7-20）。乱数を使って速度や色にある程度の幅を持たせるといっそう効果的です。

List 7-3は星をスクロールさせるプログラムです。それぞれの星について座標の更新とピクセルの描画を行います。

Fig. 7-19 星のスクロール

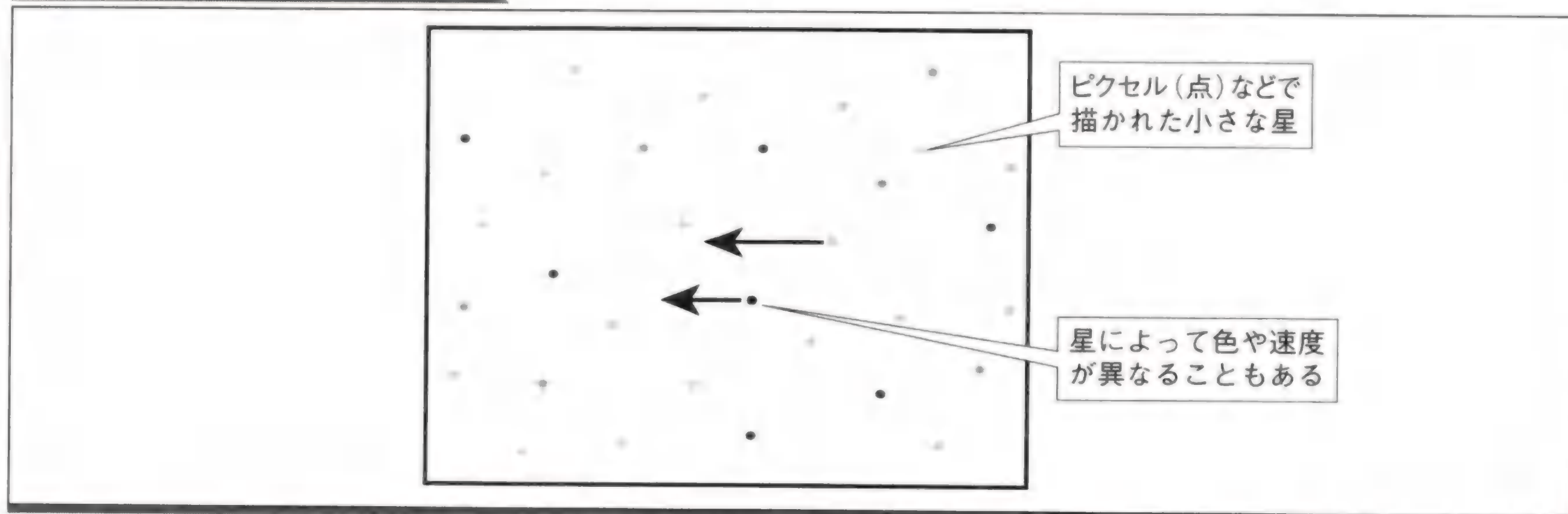
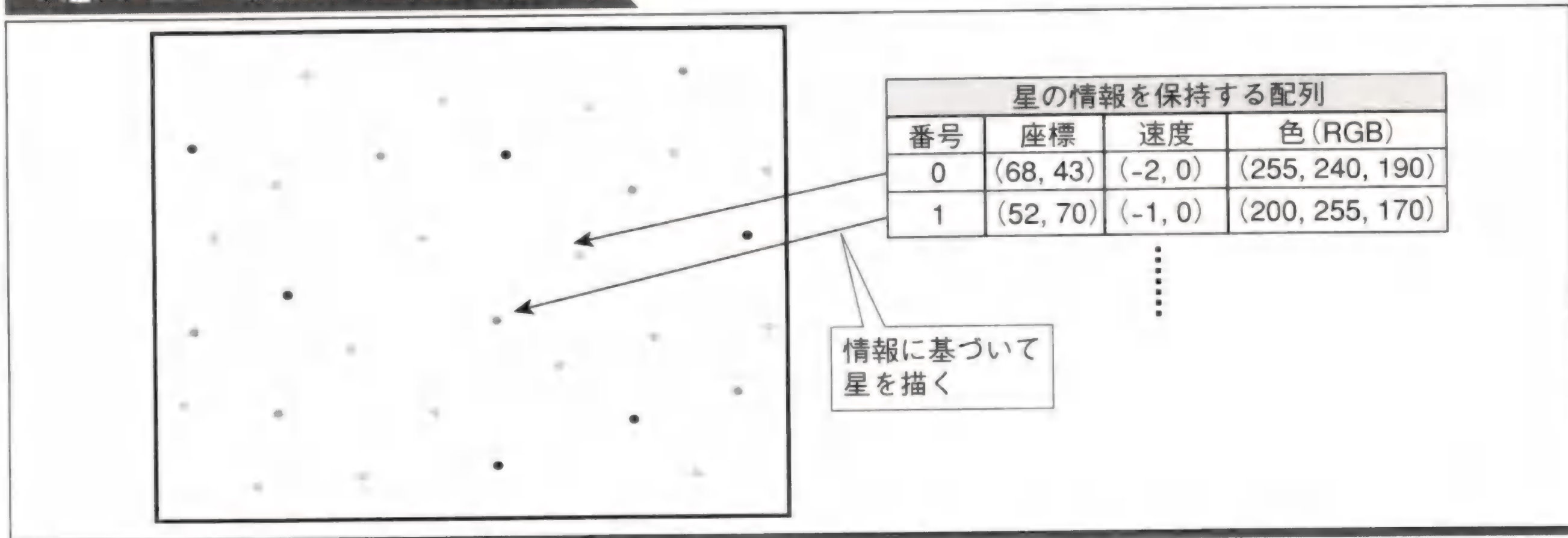



Fig. 7-20 星をスクロールさせる方法

サンプル

星のスクロール

→ P. 321

List 7-3 星のスクロール

```
// 星の情報を保持する構造体
typedef struct {
    int X, Y;     // 座標
    int VX, VY;  // 速度
    int Color;     // 色
} STAR;

// 星のスクロール
void DrawStar(
    int num_star,    // 星の数
    STAR star[]     // 星の情報
) {
    // 星の移動と描画:
    // 描画の具体的な処理はDrawPixel関数で行うとする。
    for (int i=0; i<num_star; i++) {
        star[i].X+=star[i].VX;
        star[i].Y+=star[i].VY;
        DrawPixel(star[i].X, star[i].Y, star[i].Color);
    }
}
```

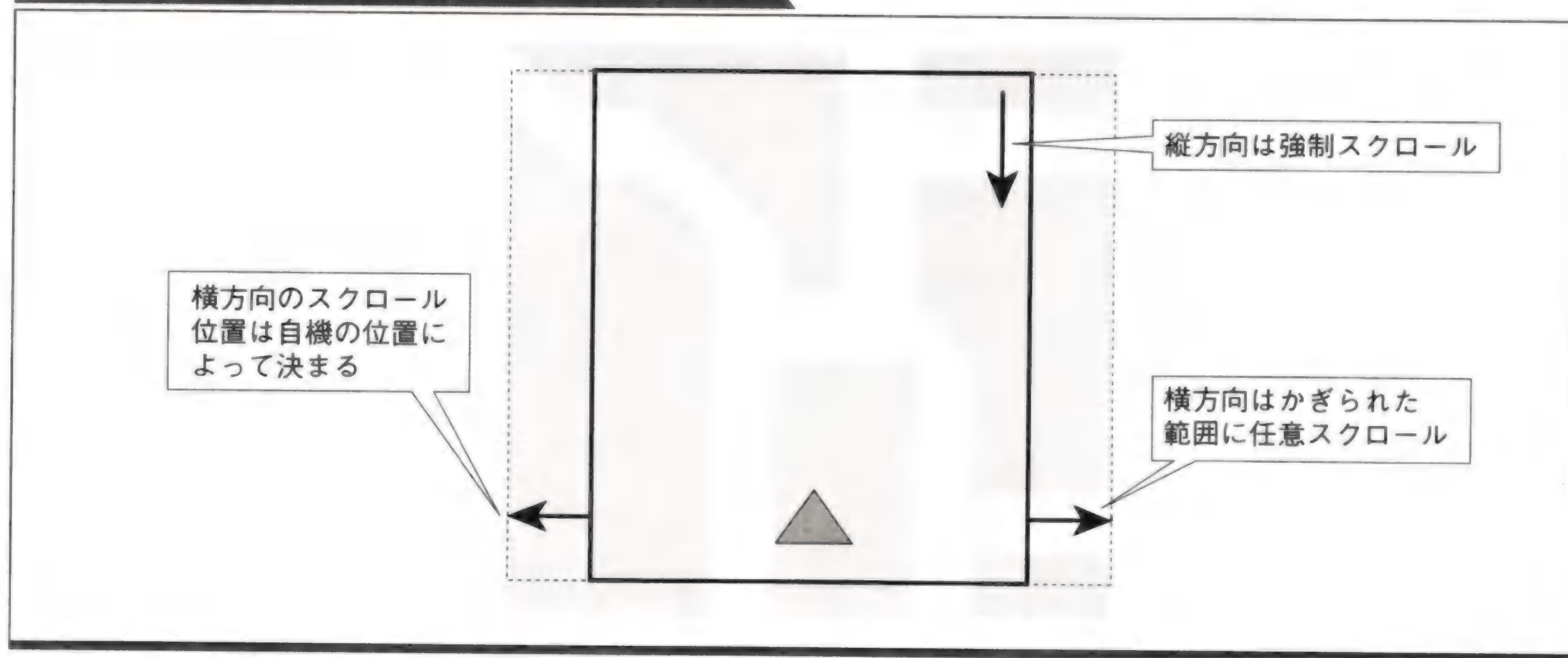
強制縦スクロール+限定横スクロール

縦方向には強制スクロールで、横方向には任意スクロールという方式です(Fig. 7-21)。この方式は多くの縦スクロールシューティングゲームに採用されています。縦方向は強制スクロールですが、画面の左右端には少しずつ動ける余地があり、自機を左右に動かすと限定的ながら任意スクロールが可能です。

この方式ではスクロールによる表示位置を自機の位置から決めます(Fig. 7-22)。自機の背景上のX座標をx、自機の幅をw、スクロール位置のX座標をsx、画面幅をsw、背景の幅をbwとし、仮にsxが、

sx=a*x+b

で表せるとします。

Fig. 7-21 強制縦スクロール+限定横スクロール

ここで、自機が画面の左右端にいるときにスクロール位置もちょうど背景の端になるとすると、次のようになります(Fig. 7-23、7-24)。

- **自機が左端にいるとき:x=0, sx=0**
- **自機が右端にいるとき:x=bw-w, sx=bw-sw**

この2つの条件を使ってaとbを求めると、

a=(bw-sw)/(bw-w)
b=0

なので、

sx=(bw-sw)*x/(bw-w)

となります。この式を使ってスクロール位置を決めれば、自機の移動に従って画面を左右にスクロールさせることができます。

注意が必要なのは、上記の式における自機のX座標は画面上のX座標ではなく、背景(マップ)上のX座標だということです。画面上の自機の座標(自機を描画するときの座標)は、x-sxのように、スクロール位置からの相対位置にする必要があります。こういった「スクロール位置からの相対位置にキャラクターを表示する」という処理は、画面がスクロールするゲームにはつきものです。

なお、xが画面上のX座標を表すときには、sxは次の式で求められます。

sx=(bow-sw)*x/(sw-w)

List 7-4は強制縦スクロール+任意横スクロールの処理をまとめたプログラムです。

サンプル

強制縦スクロール+限定横スクロール → P. 322

Fig. 7-22 自機の位置とスクロールの位置との関係

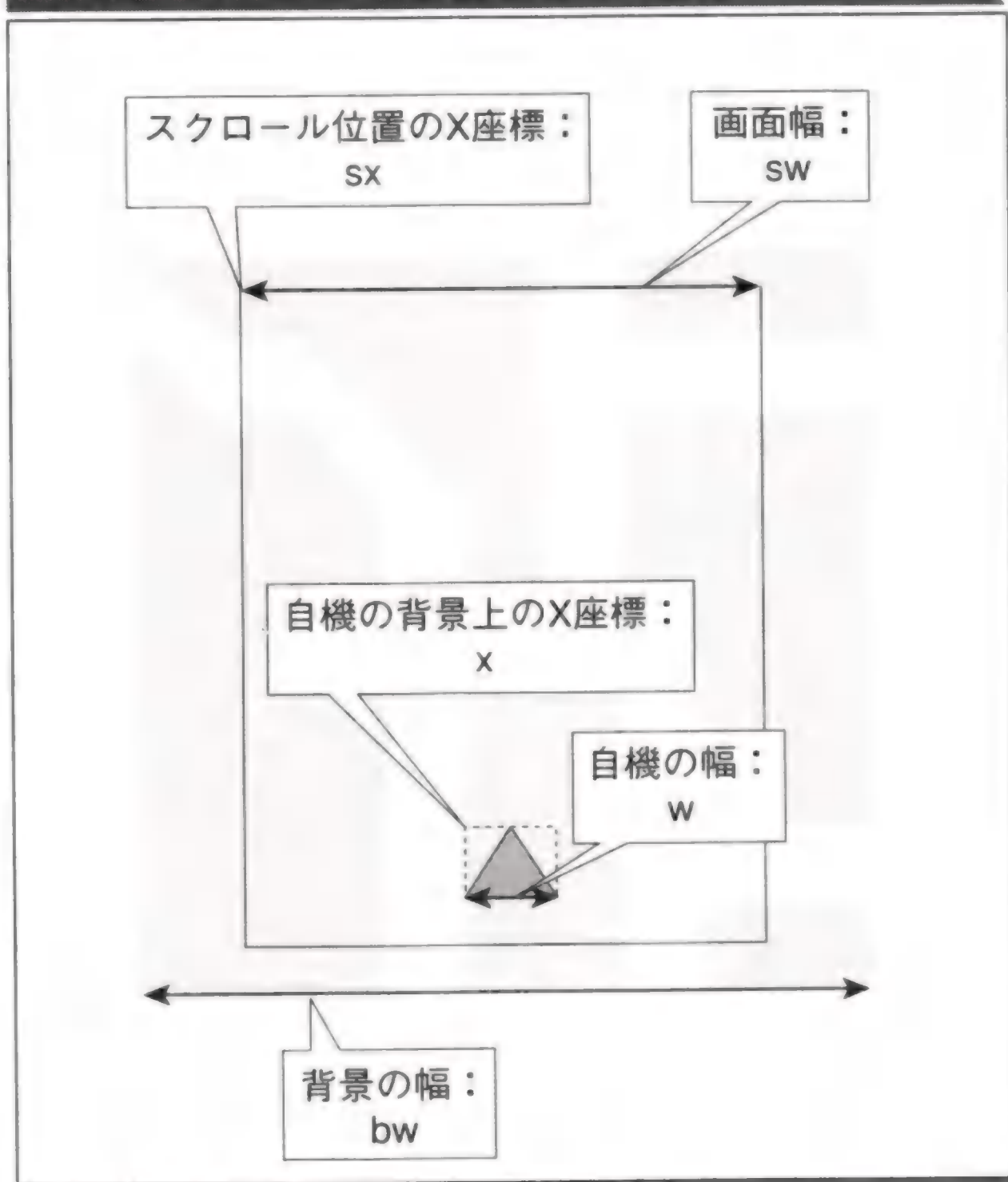



Fig. 7-23 自機が左端にいるとき

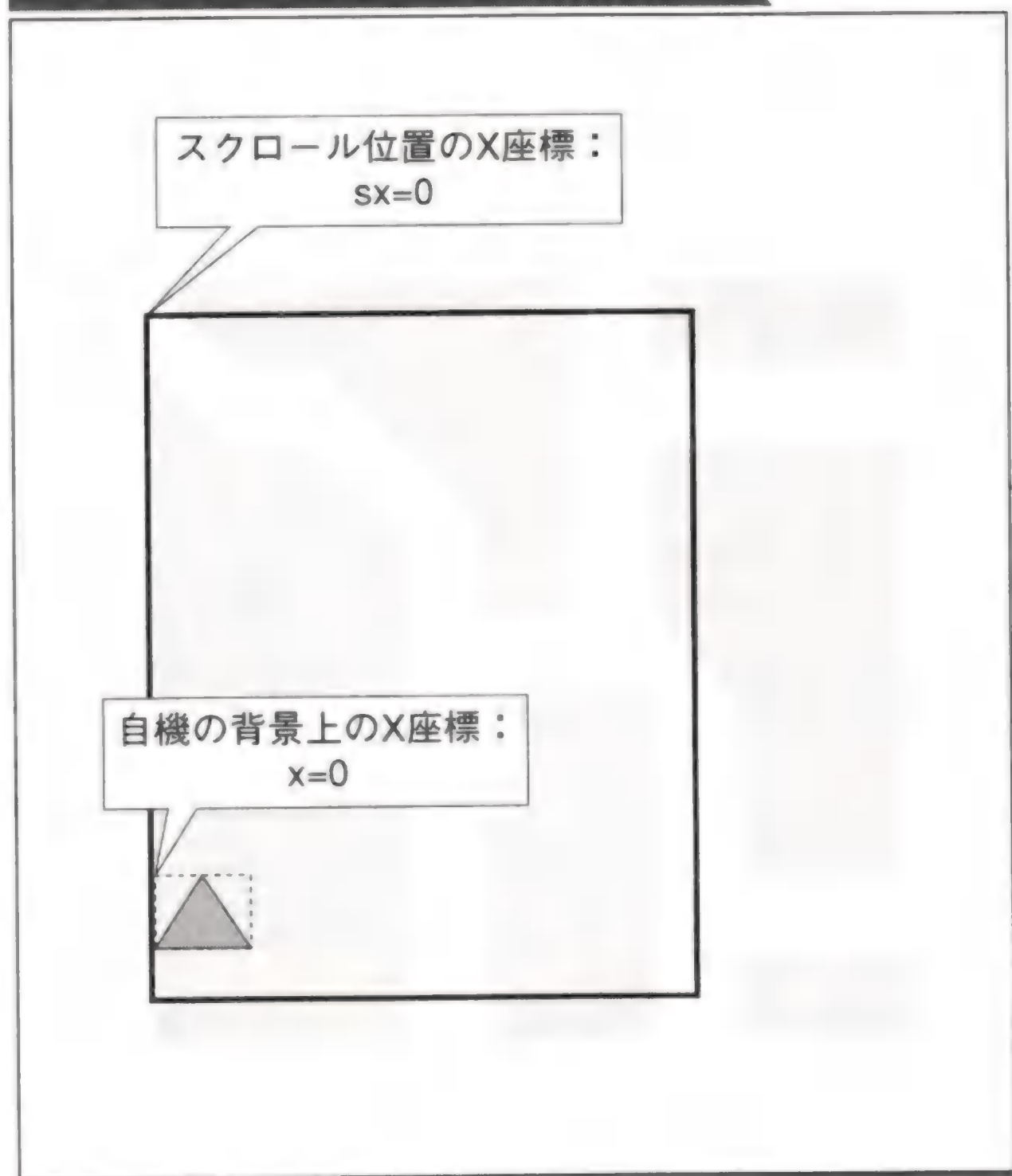
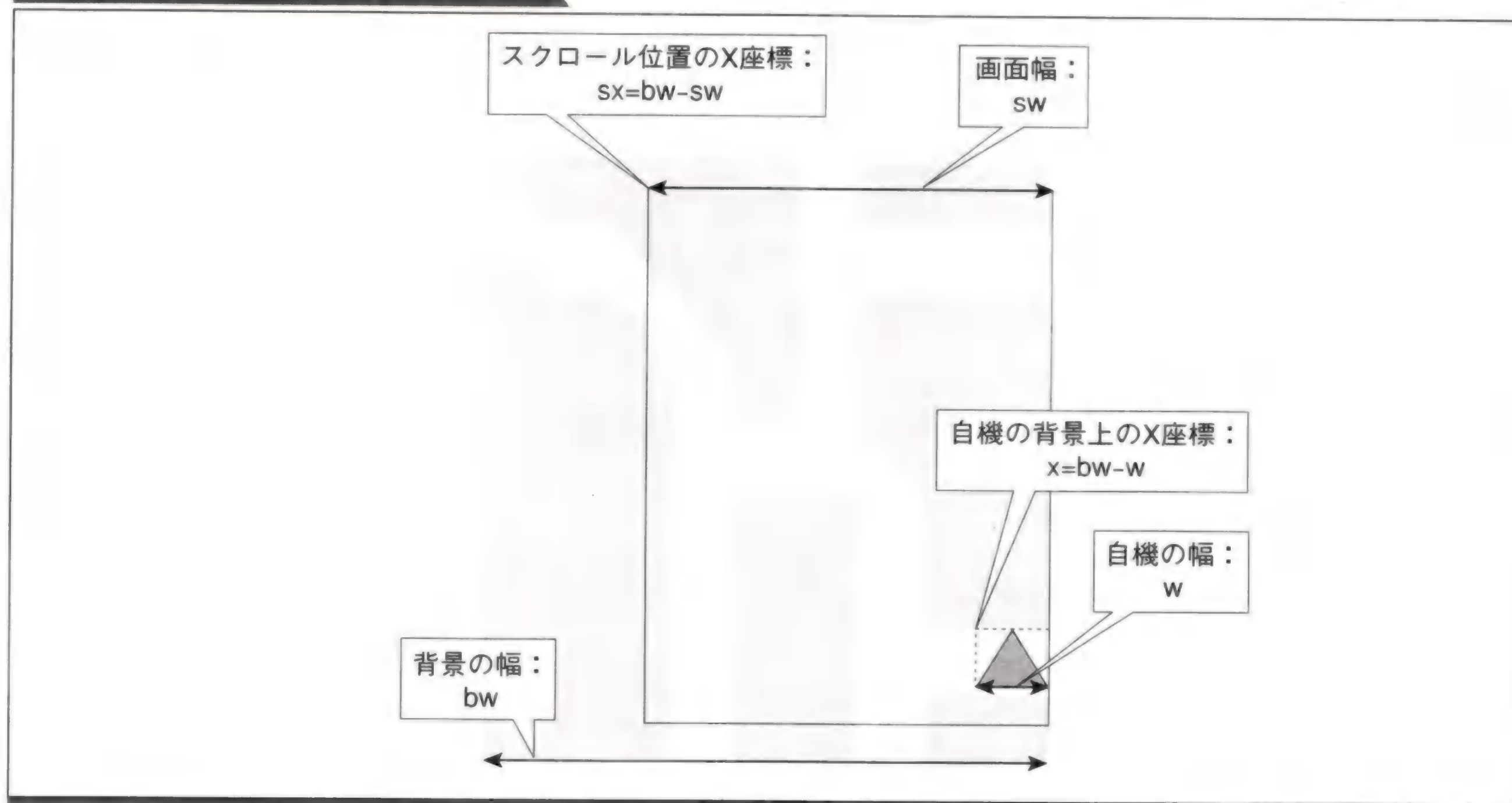



Fig. 7-24 自機が右端にいるとき

List 7-4 強制縦スクロール＋限定横スクロール

```
void MixedScroll(
    int x, int y,  // 自機の背景上の座標
    int w,         // 自機の幅
    int sy,        // スクロール位置のY座標
    int svy,       // Y方向のスクロール速度
    int sw,        // 画面幅
    int bw         // 背景の幅
) {
    // 背景の描画：
    // スクロール位置を更新し、背景を描画する。
    // 描画の具体的な処理はDrawBG関数で行うとする。
    int sx=(bw-sw)*x/(bw-w);
    sy+=svy;
    DrawBG(sx, sy);

    // 自機の描画：
    // スクロール位置からの相対位置に表示する。
    // 描画の具体的な処理はDrawMyShip関数で行うとする。
    DrawMyShip(x-sx, y-sy);
}
```

強制横スクロール+任意縦スクロール

横方向には強制スクロールで、縦方向には任意スクロールという方式です。例としては「グラディウスII（→ P. 326）」の1面などがあります（Fig. 7-25）。「グラディウスII」の場合、横方向へは強制スクロールですが、縦方向はプレイヤーが自由にスクロールさせることができます。

「グラディウスII」の場合、縦スクロールの範囲は制限されておらず、どこまでもスクロールを続けることができます。実はこれは、背景の上下がつながっているのです（Fig. 7-26）。上下どちらかの端に達すると、自機は逆の端から出てきます。

このように上下がつながった背景を表現する場合、背景の表示方法に注意が必要です（Fig. 7-27）。基本的には「背景の表示」（→ P. 270）で解説した方法でよいのですが、表示部分が上下の端にかかっているときには、上端と下端の両方を表示しなければなりません。

List 7-5は上下左右がつながった背景を表示するプログラムです。上から下に向かって背景を描きますが、描画の途中で下端に達したときに上端に戻るよう、ループの処理を工夫しています。左右に関しても同様です。

サンプル

強制横スクロール+任意縦スクロール → P. 322

Fig. 7-25 強制横スクロール+任意縦スクロール

Fig. 7-26 上下がつながった背景

表示範囲は
上下に移動する

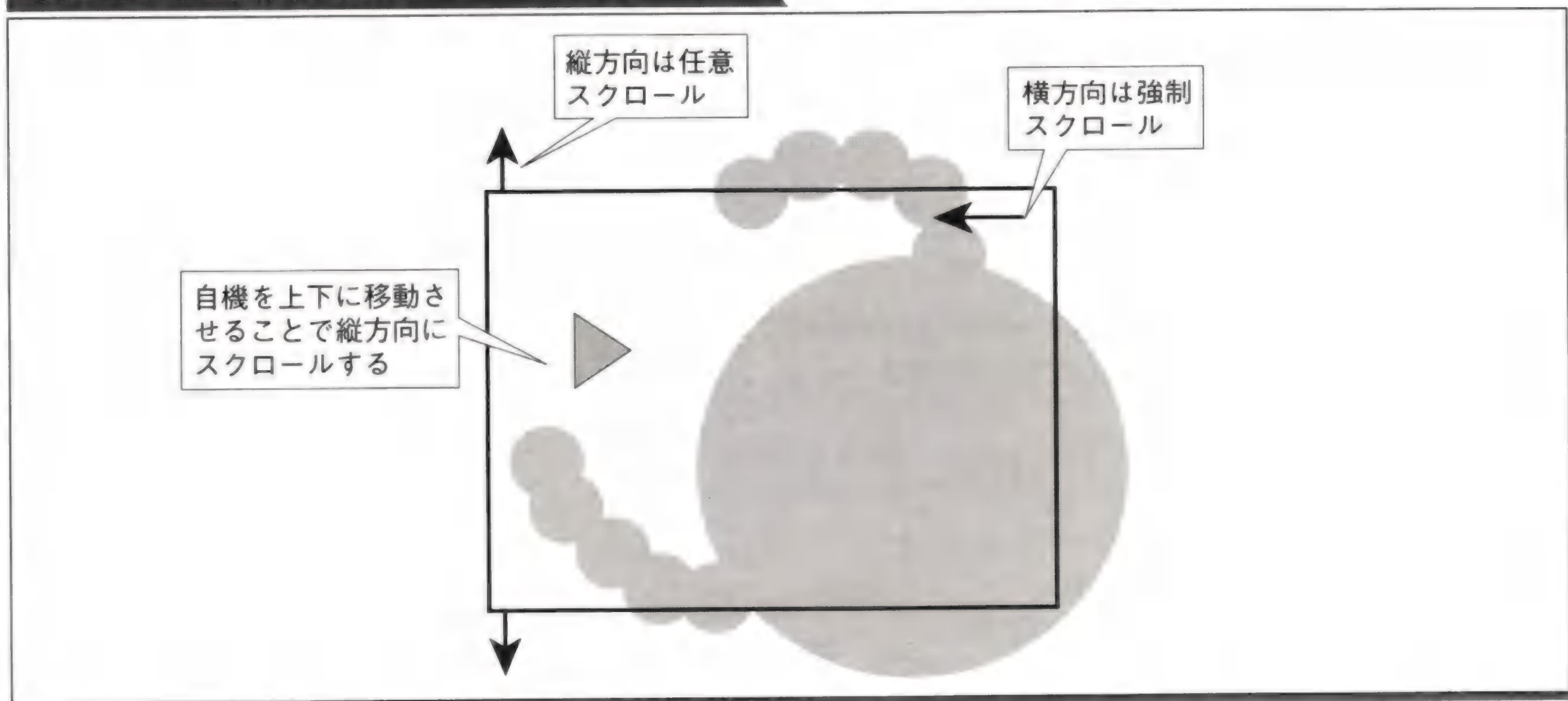
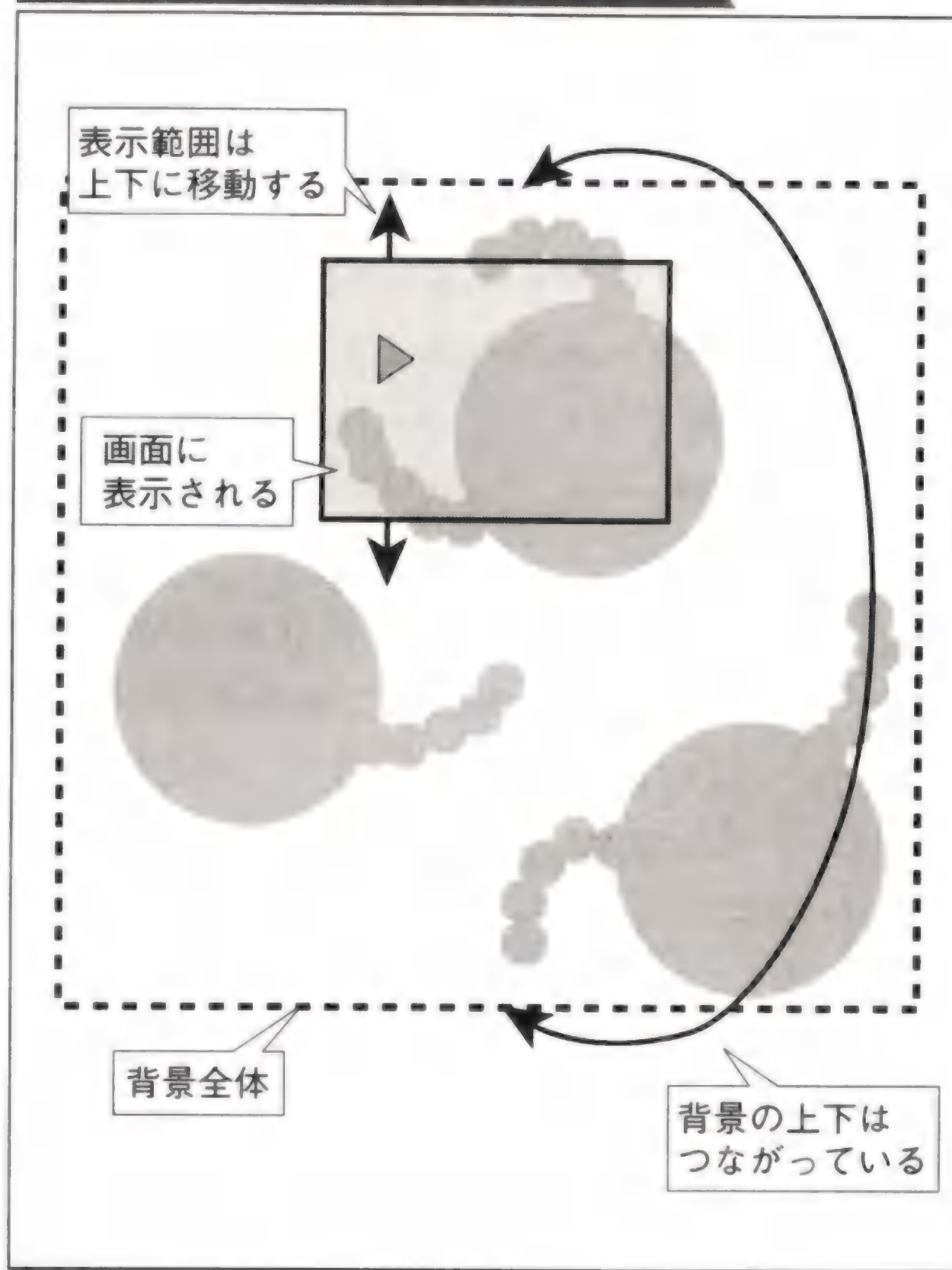
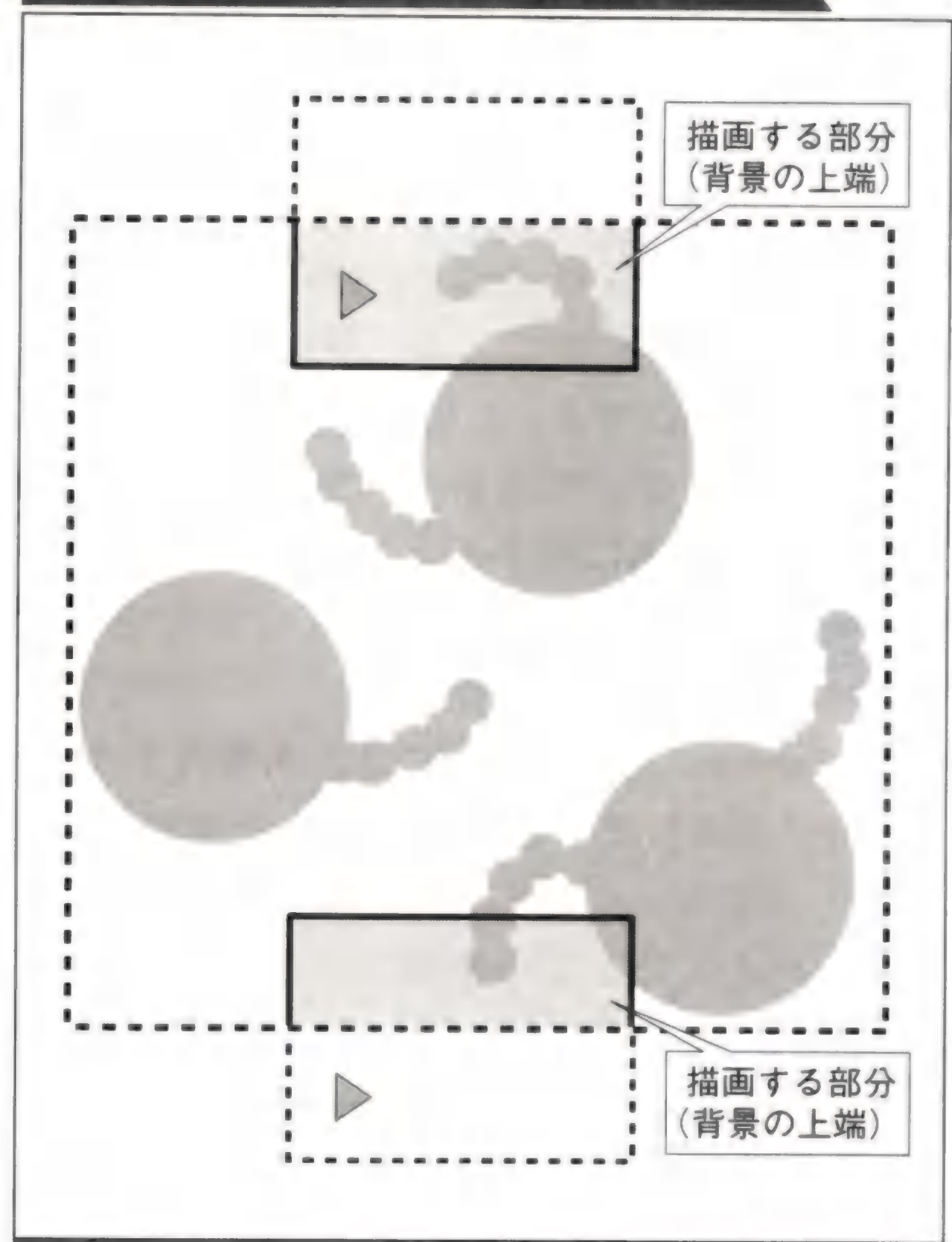
画面に
表示される

背景全体

背景の上下は
つながっている

Fig. 7-27 上下がつながった背景の表示

描画する部分
(背景の上端)

描画する部分
(背景の上端)

List 7-5 上下左右がつながった背景の表示

```
// チップ数(X方向、Y方向)
#define XMAX 1000
#define YMAX 50

// 上下左右がつながった背景の表示
void DrawCirculatedBG(
    int sx, int sy,       // 画面左上に対応する背景上の位置
    int sw, int sh,       // 画面の大きさ
    int cw, int ch,       // チップの大きさ
    int map[XMAX][YMAX]   // 背景データ(チップ番号の配列)
) {
    // 描画するチップの範囲
    int x0=sx/cw, y0=sy/ch;      // 左上端のチップ
    int x1=(sx+sw-1)/cw%XMAX,
        y1=(sy+sh-1)/ch%YMAX;   // 右下端のチップ

    // チップの描画:
    // 各チップの描画はDrawChip関数で行うとする。
    for (int i=x0; i!=x1+1; i=(i+1)%XMAX) {
        for (int j=y0; j!=y1+1; j=(j+1)%YMAX) {
```

```
            DrawChip(map[i][j], i*cw-sx, j*ch-sy);
        }
    }
}
```

回転

画面を回転させる演出です（Fig. 7-28）。回転を使ったゲームはそれほど多くありませんが、ナムコの「システムⅡ」アーケード基板は拡大縮小機能や回転機能を備えていたので、これらの機能を使って「アサルト（→ P. 323）」や「フェリオス（→ P. 333）」といったゲームが作られました。「アサルト」の場合は背景が自機の回転に合わせて回転する「任意回転」で、「フェリオス」の場合は背景が勝手に回転する「強制回転」です。

画面が回転したときには、画面内に入る背景の範囲が斜めになります（Fig. 7-29）。この斜めの範囲を厳密に計算するのはめんどうですが、表示される可能性のある最大の範囲を計算するのは簡単です（Fig. 7-30）。

Fig. 7-28 回転

Fig. 7-29 回転時に画面内に入る背景の範囲

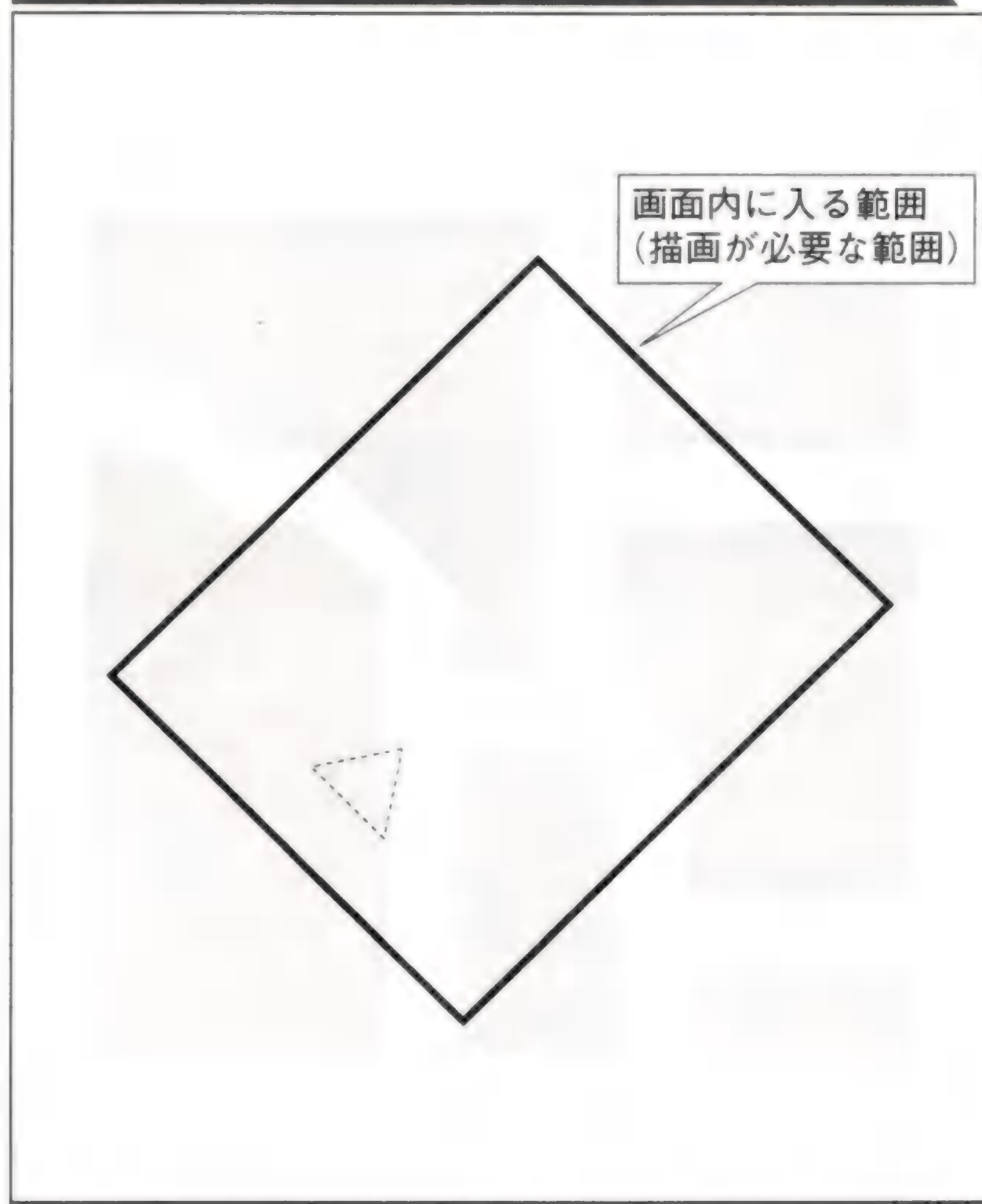
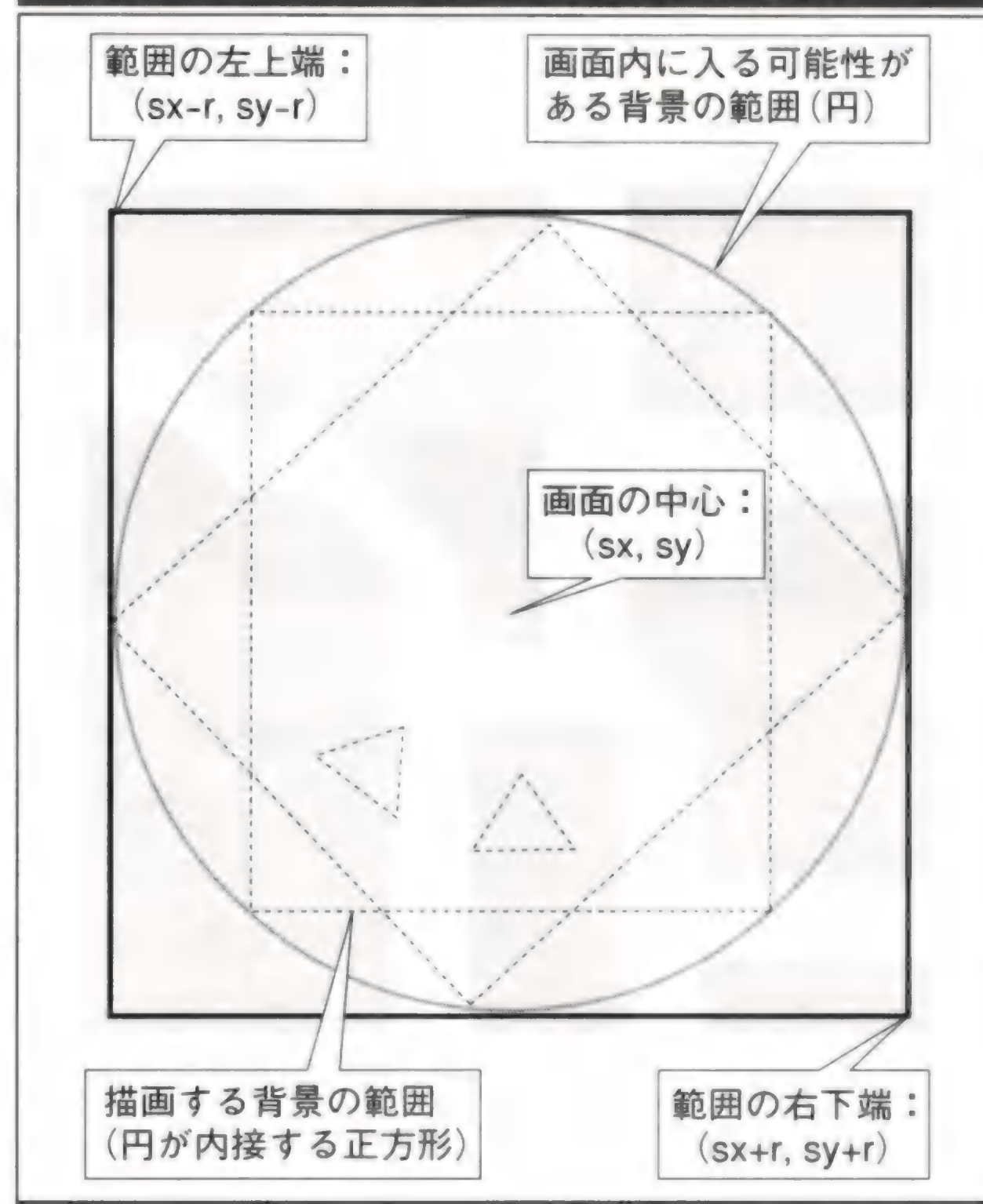



Fig. 7-30 回転時に画面内に入る背景の最大範囲

画面の中心を回転の中心とし、画面の幅の1/2をw2、高さの1/2をh2とすると、回転させた画面は半径sqrt (w2*w2+h2*h2) の円のなかに収まります。したがって、この円が内接する矩形部分の背景を描画すれば、画面が回転した場合にもきちんと背景が表示されることになります。

ここで円の半径を、

r=sqrt (w2*w2+h2*h2)

とし、画面の中心座標を (sx, sy) とすると、矩形の範囲は次のようになります。

(sx-r, sy-r) ～ (sx+r, sy+r)

この範囲の背景を「背景の表示」(→ P. 270) と同じ方法で描いてから、その描画結果を回転表示すればよいのです。

回転表示については、昔は回転専用のハードウェアを使いましたが、現在では3Dグラフィック機能を使うのが便利です。この場合には、画面が回転するように視点 (カメラ) を設定しておいて、先に説明したような範囲の背景を描画します。List 7-6は回転用に背景を描画するプログラムです。

サンプル

回転 → P. 322

List 7-6 回転

```
#include <math.h>

// チップ数(X方向、Y方向)
#define XMAX 20
#define YMAX 400

// 背景の表示
void DrawBG(
    int sx, int sy,       // 画面の中心座標
    int sw, int sh,       // 画面の大きさ
    int cw, int ch,       // チップの大きさ
    int map[YMAX][XMAX]   // 背景データ(チップ番号の配列)
) {
    // 回転で画面内に入る背景の最大範囲を求める
    int w2=sw/2, h2=sh/2;                // 画面の幅と高さの1/2
    int r=(int)sqrt(w2*w2+h2*h2);        // 円の半径
    int x0=(sx-r)/cw, y0=(sy-r)/ch;      // 左上端のチップ
    int x1=(sx+r)/cw, y1=(sy+r)/ch;      // 右下端のチップ

    // チップの描画:
    // 各チップの描画はDrawChip関数で行うとする。
    for (int i=y0; i<=y1; i++) {
        for (int j=x0; j<=x1; j++) {
            DrawChip(map[i][j], j*cw-sx+sw/2, i*ch-sy+sw/2);
        }
    }
}
```

高速スクロール

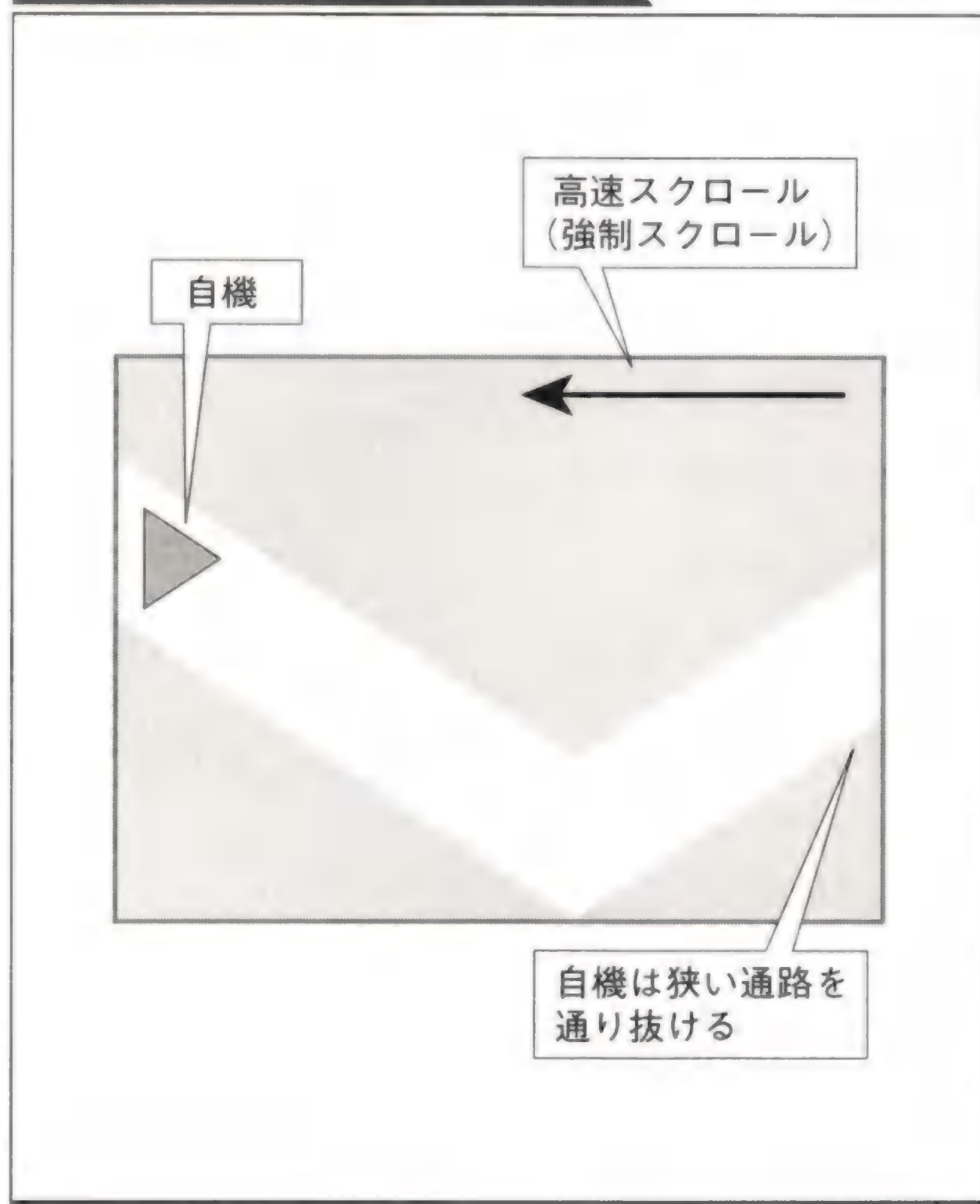
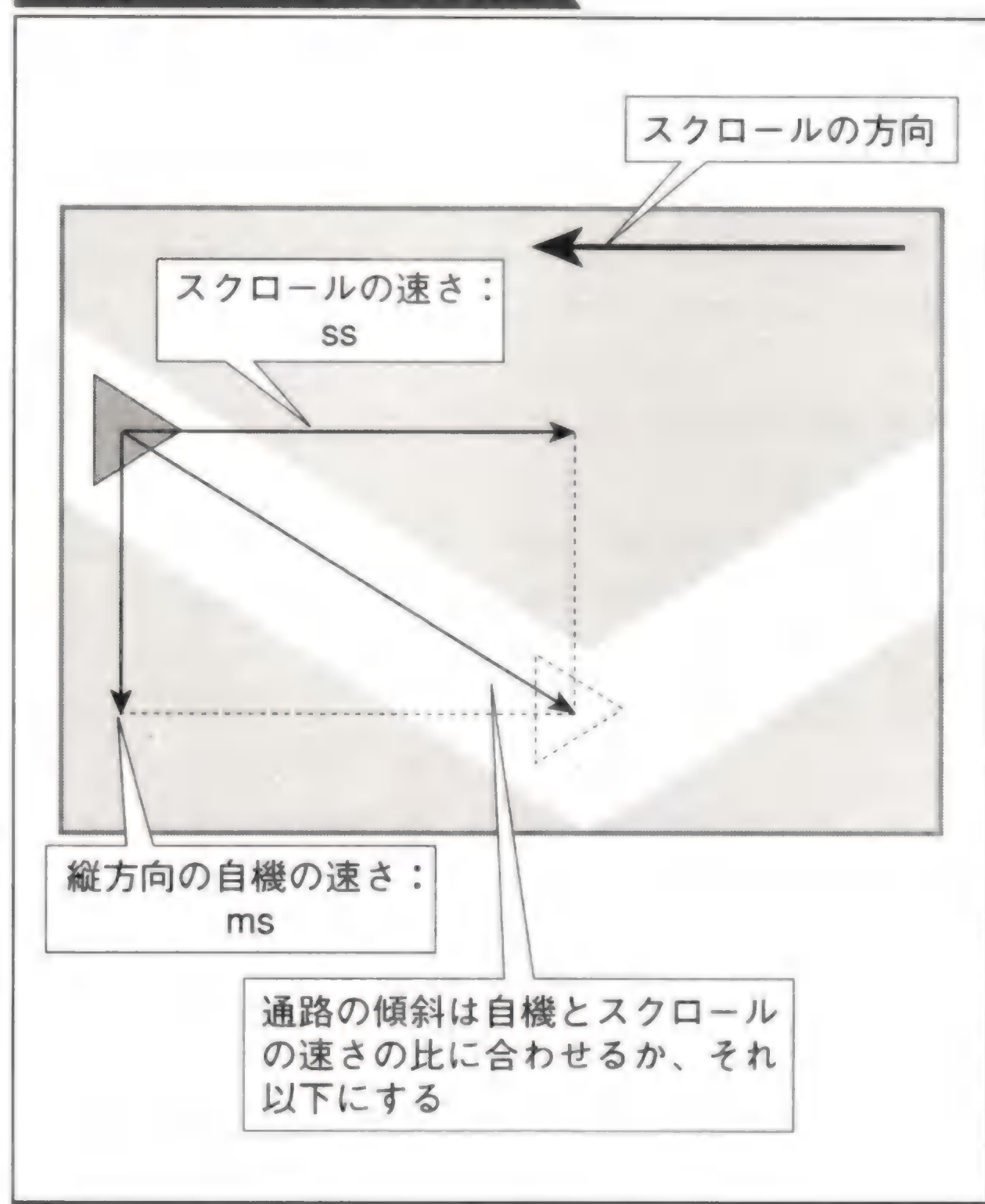
画面を高速にスクロールさせる演出です(Fig. 7-31)。迷路状になった狭い通路を通り抜けさせる際に使うことが多いのですが、単に飛翔感を演出したい場合にも使うことができます。高速スクロールといっても処理のうえでは特別なことはなく、単にスクロールのスピードを速くするだけです。ただし、斜めに傾斜した通路を作る場合には、スクロールの速さと自機の速さの比を考慮して、通路の傾斜を決める必要があります(Fig. 7-32)。

自機が通路を抜けられるようにするには、自機の速さをms、スクロールの速さをssとすると、通路の傾斜(横方向の移動量に対する縦方向の移動量)をss/ms以下にする必要があります。長い通路では、傾斜をちょうどss/msに合わせるのもよい方法です。こうすればスティックを一方向に入れっぱなしにするだけで通路を抜けることができるので、爽快感が生まれます。

Fig. 7-31 高速スクロール

高速スクロール
(強制スクロール)
自機
自機は狭い通路を
通り抜ける

Fig. 7-32 通路の傾斜

スクロールの方向
スクロールの速さ：
ss
縦方向の自機の速さ：
ms
通路の傾斜は自機とスクロール
の速さの比に合わせるか、それ
以下にする

自機に力をかける

スクロールの速さを表現するために、自機に力（慣性）をかけるという演出もあります。たとえば上から下への高速スクロール中に、自機に対して下方向の力をかけることによって、スクロールの速さを強調します。

力をかけるといっても方法は簡単で、スクロールと同じ方向に自機の速度を変化させてやるだけです（Fig. 7-33）。たとえば自機の速さが2で、力による速度の変化が下方向に1だとすると、

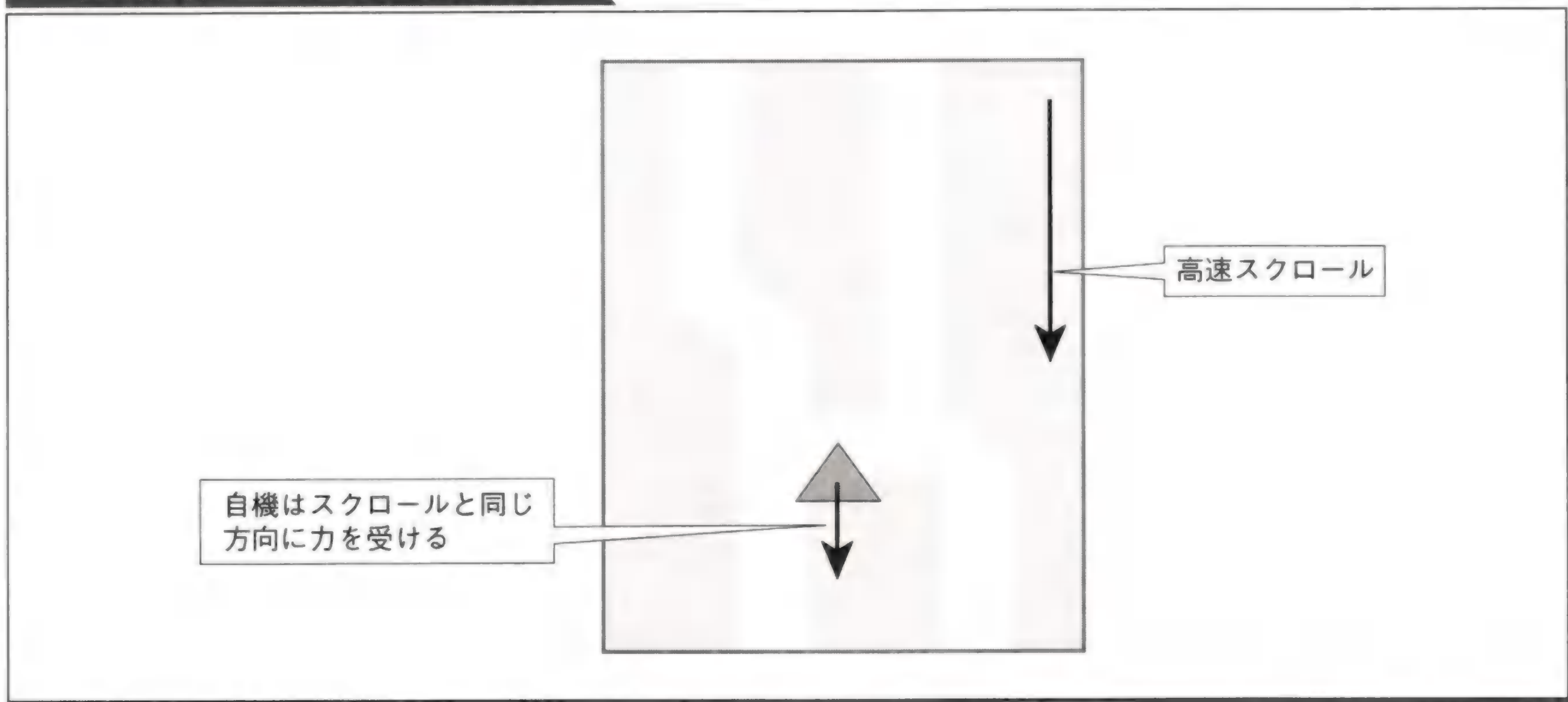
- **自機が上に移動する速さ：2−1=1**
- **自機が下に移動する速さ：2+1=3**

となり、自機はあたかも下方向に力を受けているような動きをします。

List 7-7は自機にスクロールと同じ方向の力をかけるプログラムです。ここではスクロール速度の半分を自機の速度に加えています。

サンプル

高速スクロール → P. 232

List 7-33 自機の速度を変化させる

List 7-7 スクロールと自機に対する力

```
void ScrollForce(
    int& x, int& y,    // 自機の座標
    int vx, int vy,    // 自機の速度
    int svx, int svy   // スクロールの速度
) {
    x+=vx+svx/2;
    y+=vy+svy/2;
}
```

Stage＃90
高速スクロール

プレイヤーによるスクロール速度の調節

基本的には強制スクロールながらも、スクロールの速度はプレイヤーがある程度調節できるというものです。これはあまり見かけないルールですが、「疾風魔法大作戦（→ P. 328）」ではこのルールを使ってレースゲーム的な要素をシューティングゲームに盛り込んでいます。

「疾風魔法大作戦」の場合、自機が画面の上方にいるとスクロールのスピードが上がり（Fig. 7-34）、画面の下方にいるとスピードが下がります（Fig. 7-35）。敵は画面の上方からくるので自機は画面の下方にいるのが安全なのですが、このゲームには「生き残る」のと同時に「レースに勝つ」という目的が設けられています。レースに勝つには、積極的に危険な画面の上方にいってスクロールのスピードを上げ、速くゴールに到達する必要があるのです。

Fig. 7-34 自機が画面上方にいる場合

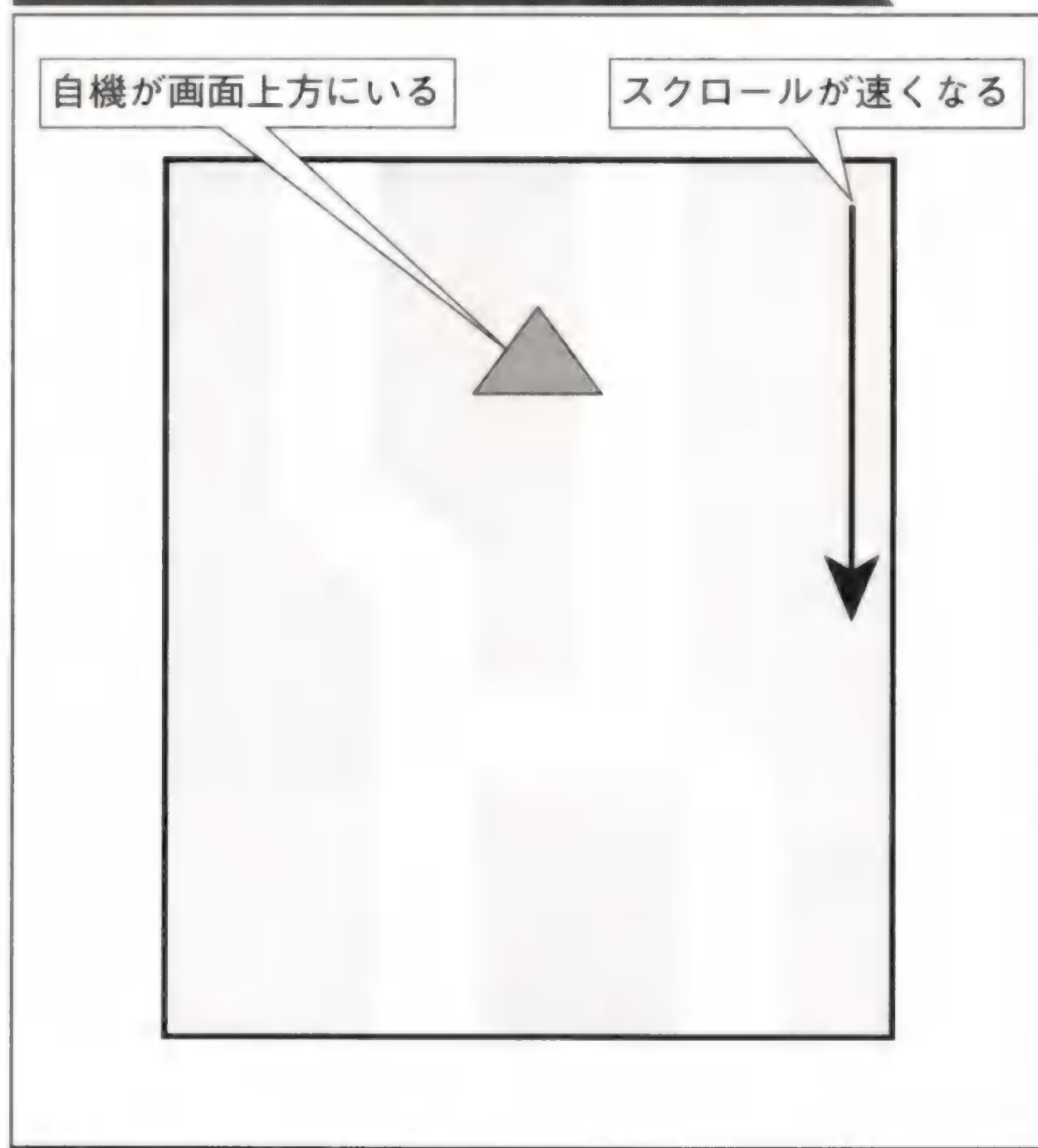
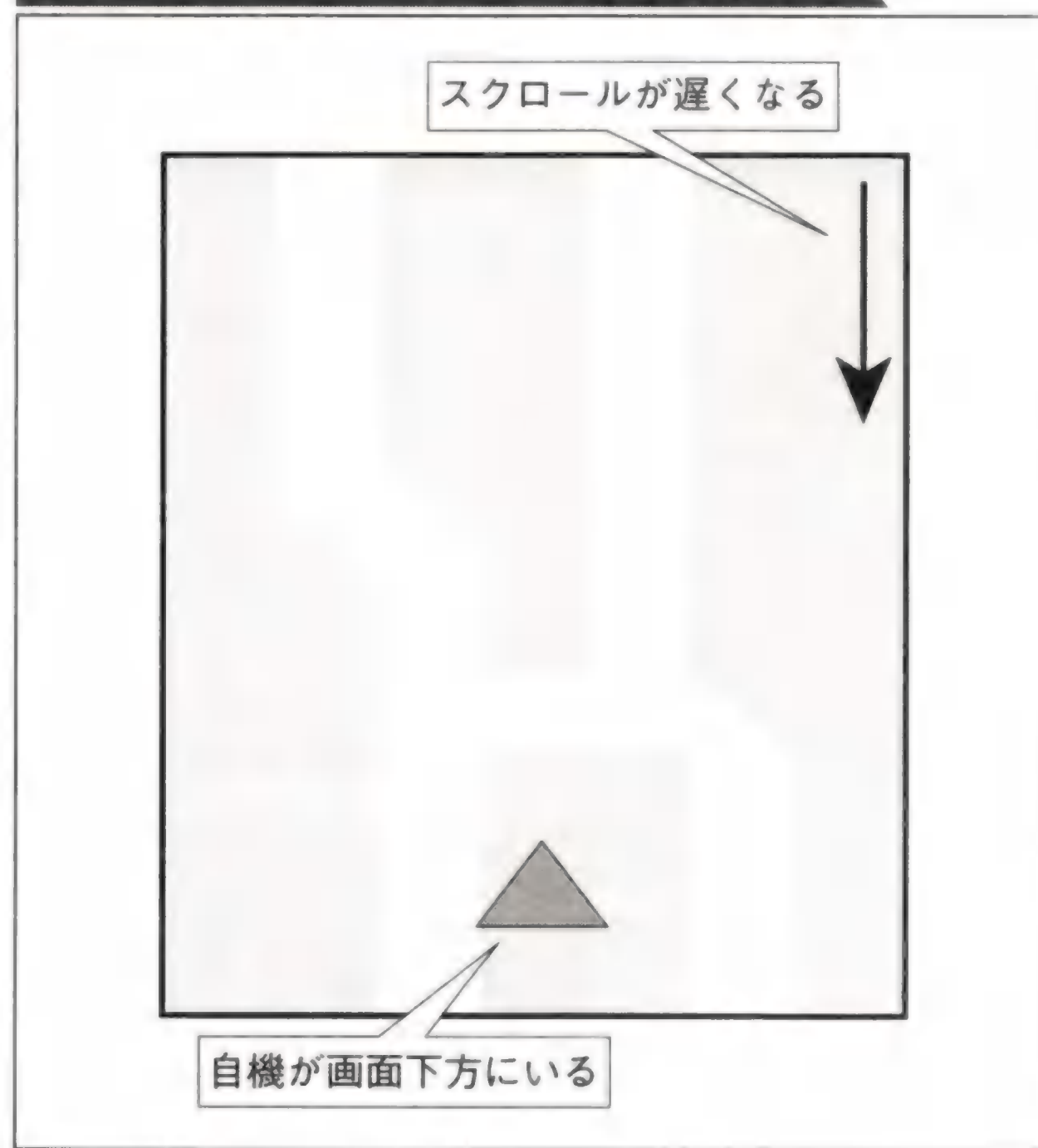



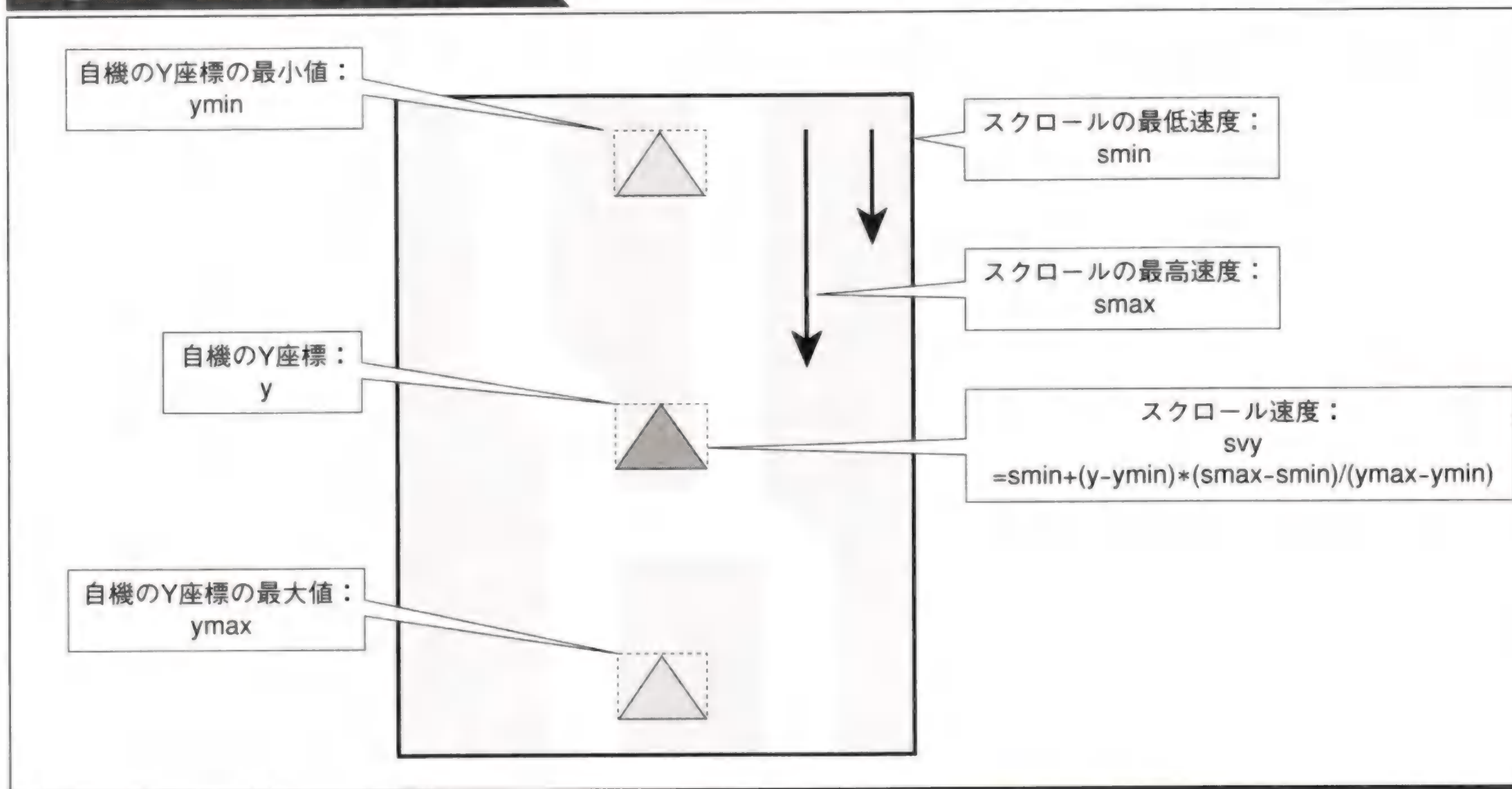
Fig. 7-35 自機が画面下方にいる場合

自機の位置に応じたスクロール速度の計算方法は次のとおりです（Fig. 7-36）。ここでは、自機の位置によってスクロール速度がなめらかに変わるようにします。自機のY座標をy、Y座標の最小値（画面の上方）をymin、最大値（画面の下方）をymax、スクロールの最大スピードをsmax、最小スピードをsminとすると、スクロールの速度svyは次の式で計算できます。

svy=smin+ (y-ymin) * (smax-smin) / (ymax-ymin)

List 7-8は自機の位置からスクロール速度を計算するプログラムです。

サンプル

プレイヤーによるスクロール速度の調節 → P. 322

Fig. 7-36 スクロール速度の計算

List 7-8 スクロール速度の計算

```
void ScrollSpeed(
    int y,                  // 自機のY座標
    int& svy,               // スクロールの速度(Y方向)
    int ymin, int ymax,     // 自機Y座標の最大値、最小値
    int smax, int smin      // スクロールの最大、最小スピード
) {
    svy=smin+(y-ymin)*(smax-smin)/(ymax-ymin);
}
```

Stage 7のまとめ ▶▶

多くのシューティングゲームでは、各ステージがそれぞれ異なるテーマに基づいてデザインされています。たとえば宇宙のステージがあったり、砂漠や水中のステージがあったりといった具合です。そしてステージのテーマによって、背景や地形のデザインも変わります。たとえば宇宙ステージには彗星、砂漠ステージには流砂、水中ステージには海流や泡などがあります。プレイヤーにとっては、ステージによってさまざまに変わる背景を見るのもゲーム攻略の楽しみです。

ということで、「背景はゲームの雰囲気を高めるとともに、プレイヤーにゲームを遊び込ませる『餌』でもある！」というのが本章のまとめです。

システム

System

これまでの章で解説した弾、自機、武器、特殊攻撃、敵、背景があれば、シューティングゲームに必要な要素はほとんど揃ったことになります。あとの残りはゲームシステム全体に関する機能です。

システム全体に関する事柄としては、スコア、ボーナス、リプレイ、難易度などがあります。ほかの要素に比べると地味な部分ではありますが、こういった部分のしあがりによってゲーム全体の面白さが大きく変わってくるのです。

桁数の多いスコア

ここでは何億、何兆という大きなスコアを扱う方法を解説します。最近のゲームにはスコアの上限が非常に高いものがありますが、こういったスコアは普通の整数演算では扱えないことがあります。

たとえば「ギガウィング（→ P. 325）」シリーズはスコアが「億」の単位を超えて「京（けい）」や「亥（がい）」の単位にまで達します。1億、1京、1亥はそれぞれ次のような桁数になります。

- **1億 =　　　　　　100,000,000（0が8個）**
- **1京 =　　　1,000,000,000,000（0が12個）**
- **1亥 = 10,000,000,000,000,000（0が16個）**

それに対して、プログラミング言語で使う一般的な32ビット整数（int型など）が表せる数値の範囲は次のとおりです。

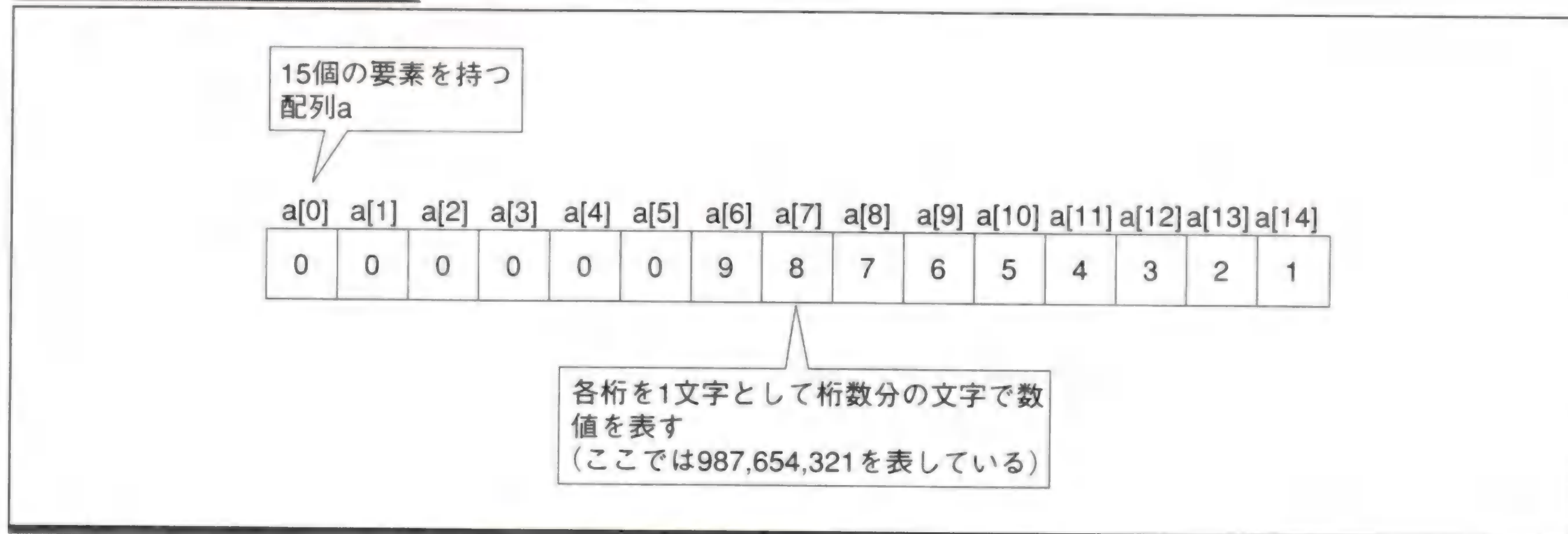
-2,147,483,648 ～ 2,147,483,647

これでは、せいぜい20億までしか表せません。符号なし32ビット整数（unsigned int型など）を使ったとしても、表せるのは40億までです。一方、プログラミング言語によっては64ビット整数が使えます。64ビット整数が表せる数値の範囲は次のとおりです。

-9,223,372,036,854,775,808 ～ 9,223,372,036,854,775,807

これで900亥まで大丈夫です。符号なし64ビット整数なら、ほぼ倍の1,800亥まで表せます。

このように64ビット整数を使えばたいていのスコアは扱えますが、64ビット整数が使えないプログラミング言語もありますし、もしかするとゲームによってはさらに大きなスコアを扱う必要があるかもしれません。そういった場合には、既存の32ビット整数や64ビット整数は使わずに、「多倍長演算」という手法を用います（Fig. 8-1）。

Fig. 8-1 多倍長演算

多倍長演算では、数値の各桁を1文字（char型など）として、桁数分の文字を並べて数値全体を表現します。文字を配列（char型の配列など）で保存することにして、必要な桁数だけ配列の要素を用意すれば、どんなに大きな数値でも表すことができます。

科学技術計算などに使う多倍長演算では、1桁を1文字で表すのではなく、より多くの桁を1つの数値で表すことによって処理を高速化します。ゲームの場合にはスコアを頻繁に表示する関係上、本書で解説するように1桁を1文字で表したほうが好都合です。

問題は加算や乗算といった計算です。多倍長演算でスコアを扱う場合には、最低でも次のような計算が必要になります。

- **初期化**
- **加算（多倍長＋整数）**
- **加算（多倍長＋多倍長）**
- **乗算（多倍長×整数）**

以下では、各計算方法について解説します。

■ 初期化

多倍長数を整数で初期化します。ここでは15桁まで表せる配列aに、整数87,654,321を代入することを考えてみます（Fig. 8-2-①）。

Fig. 8-2 初期化の例

① 初期状態

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	整数
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	87,654,321

② 下から1番目の桁の処理

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	整数
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	8,765,432

③ 下から2番目の桁の処理

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	整数
0	0	0	0	0	0	0	0	0	0	0	0	0	2	1	876,543

④ 終了状態

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	整数
0	0	0	0	0	0	0	8	7	6	5	4	3	2	1	0

最初は一番下の桁（配列の末尾）から始めます。代入する整数を10で割った余りを求めて、一番下の桁に代入します。また、整数は10で割ります。ここではa［14］が1になり、整数は8,765,432になります（Fig. 8-2-②）。

次の桁についても同様に、整数を10で割った余りを代入し、整数を10で割ります。ここでは

a［13］が2になり、整数は876,543になります（Fig. 8-2-③）。

ほかの桁についても同様に繰り返して、一番上の桁に達したら終わりです。結果として、a［7］〜a［14］に87,654,321が代入されます（Fig. 8-2-④）。

■ 加算（多倍長＋整数）

多倍長数に整数を加算します。ここでは87,654,321が代入された配列aに対して、整数12,345,679を加算することを考えます（Fig. 8-3-①）。

Fig. 8-3 多倍長＋整数の例

① 初期状態

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	t
0	0	0	0	0	0	0	8	7	6	5	4	3	2	1	12,345,679

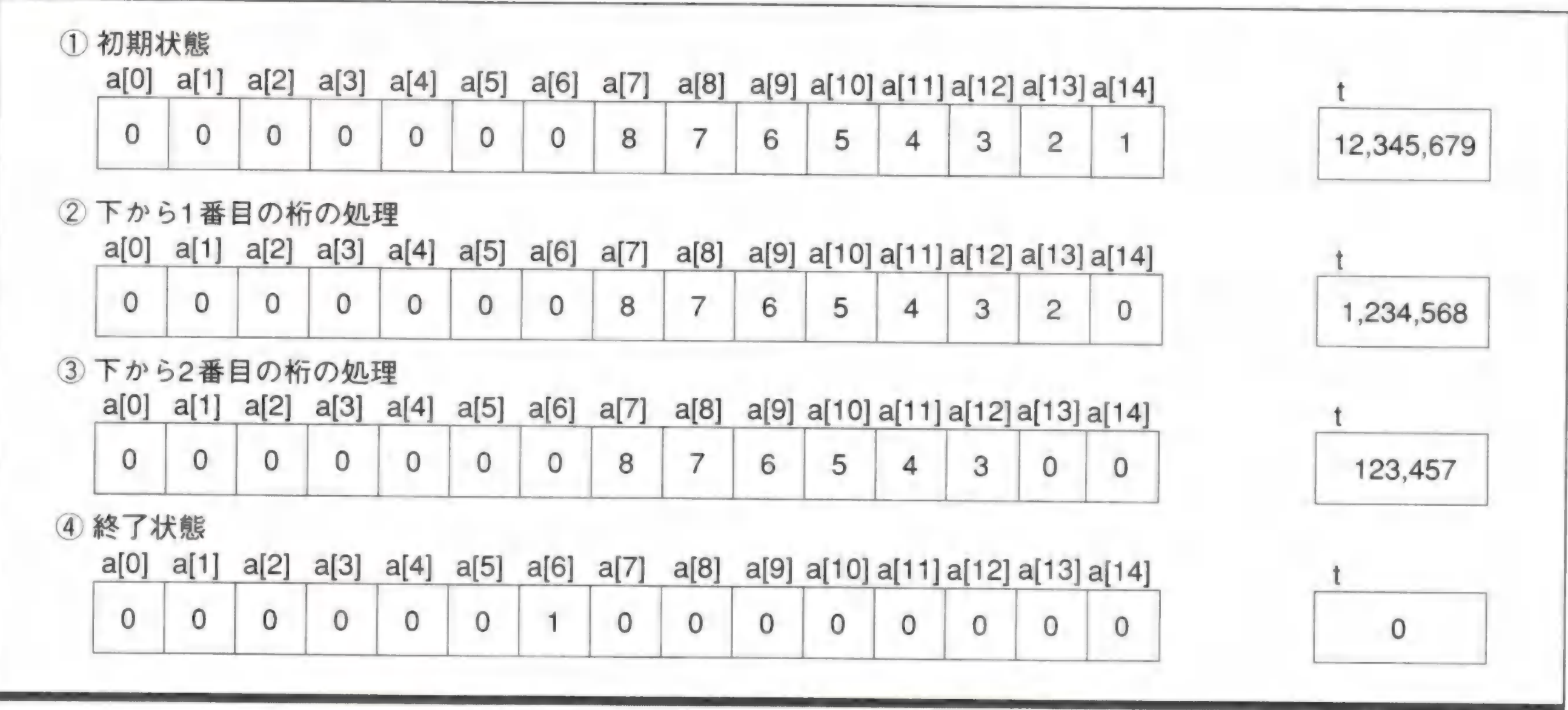
② 下から1番目の桁の処理

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	t
0	0	0	0	0	0	0	8	7	6	5	4	3	2	0	1,234,568

③ 下から2番目の桁の処理

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	t
0	0	0	0	0	0	0	8	7	6	5	4	3	0	0	123,457

④ 終了状態

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	t
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

多倍長演算での加算は、桁の大きな数の筆算をするときの方法によく似ています。この計算には結果を一時的に保存する変数tを使います。tには加算する整数を代入しておきます。

最初は一番下の桁から始めます。aの一番下の桁を取り出してtに加算し、tを10で割った余りをaに戻します。また、tを10で割ります。ここではa［14］が1、tが12,345,679なので、加算するとtは12,345,680になります。結果としてa［14］は0に、tは1,234,568となります（Fig. 8-3-②）。

次の桁についても同様です。ここではa［13］が2、tが1,234,568なので、加算するとtは1,234,570になります。結果としてa［13］は0に、tは123,457となります（Fig. 8-3-③）。

同じ処理を一番上の桁まで繰り返せば終わりです。結果は100,000,000となり、これはa［6］〜a［14］に入ります（Fig. 8-3-④）。

■ 加算（多倍長＋多倍長）

多倍長数に多倍長数を加算します。ここでは123,456,789が代入された配列aに対して、987,654,321が代入された配列bを加算することを考えます（Fig. 8-4-①）。

Fig. 8-4　多倍長＋多倍長の例

① 初期状態

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	t
0	0	0	0	0	0	1	2	3	4	5	6	7	8	9	0

b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	b[8]	b[9]	b[10]	b[11]	b[12]	b[13]	b[14]
0	0	0	0	0	0	9	8	7	6	5	4	3	2	1

② 下から１番目の桁の処理

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	t
0	0	0	0	0	0	1	2	3	4	5	6	7	8	0	1

b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	b[8]	b[9]	b[10]	b[11]	b[12]	b[13]	b[14]
0	0	0	0	0	0	9	8	7	6	5	4	3	2	1

③ 下から２番目の桁の処理

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	t
0	0	0	0	0	0	1	2	3	4	5	6	7	1	0	1

b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	b[8]	b[9]	b[10]	b[11]	b[12]	b[13]	b[14]
0	0	0	0	0	0	9	8	7	6	5	4	3	2	1

④ 終了状態

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	t
0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0

b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	b[8]	b[9]	b[10]	b[11]	b[12]	b[13]	b[14]
0	0	0	0	0	0	9	8	7	6	5	4	3	2	1

「多倍長＋整数」の場合と同じように、計算は一番下の桁から始めます。この計算には結果を一時的に保存する変数tを使います。tは0で初期化しておきます。

最初はtにaの一番下の桁とbの一番下の桁を加算します。そしてtを10で割った余りをaに戻し、tは10で割ります。ここではtが0、a［14］が9、b［14］が1なので、tは10（0+9+1）になります。したがってa［14］には0（10を10で割った余り）、tには1（10を10で割った商）を代入します（Fig. 8-4-②）。

次の桁についても同様です。ここではtが1、a［13］が8、b［13］が2なので、tは11（1+8+2）になります。したがってa［13］は1、tは1とします（Fig. 8-3-③）。

同じ処理を一番上の桁まで繰り返せば終わりです。結果は1,111,111,110となり、これはa［5］～a［14］に入ります（Fig. 8-4-③）。

■ 乗算（多倍長×整数）

多倍長数と整数の乗算です。ここでは987,654,321が代入された配列aに対して、1,000,000を

Fig. 8-5 多倍長×整数の例

① 初期状態

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	t
0	0	0	0	0	0	9	8	7	6	5	4	3	2	1	0

② 下から1番目の桁の処理

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	t
0	0	0	0	0	0	9	8	7	6	5	4	3	2	0	100,000

③ 下から2番目の桁の処理

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	t
0	0	0	0	0	0	9	8	7	6	5	4	3	0	0	210,000

④ 終了状態

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	t
9	8	7	6	5	4	3	2	1	0	0	0	0	0	0	0

乗算することを考えます（Fig. 8-5-①）。

加算の場合と同じように、多倍長演算の乗算は筆算によく似ています。この計算には結果を一時的に保存する変数tを使います。tは0で初期化しておきます。

計算は一番下の桁から始めます。aの一番下の桁を取り出して、これに整数を乗算し、結果をtに加算します。そしてtを10で割った余りをaに戻し、tは10で割ります。ここではtが0、a［14］が1、整数が1,000,000なので、tは1,000,000（0+1*1,000,000）になります。したがってa［14］は0、tは100,000とします（Fig. 8-5-②）。

次の桁についても同様です。ここではtが100,000、a［13］が2、整数が1,000,000なので、tは2,100,000（100,000+2*1,000,000）になります。したがってa［13］は0、tは210,000とします（Fig. 8-5-③）。

同じ処理を一番上の桁まで繰り返せば終わりです。結果は987,654,321,000,000となり、これはa［0］～a［14］に入ります（Fig. 8-5-④）。

※

スコア表示のために必要な多倍長演算の方法を解説しました。なお、演算結果の多倍長数をスコアとして表示するには、配列を1桁ずつ読み出して、それぞれの数値に対応した文字を表示します。

List 8-1はこれまでに解説した多倍長演算の処理をまとめたものです。このプログラムは単体で実行できます。実行結果は次のとおりです。

```
a: 87654321
a+=12345679: 100000000

a: 123456789
b: 987654321
a+=b: 1111111110
```

a: 987654321
a*=1000000: 987654321000000

List 8-1で桁数Nを増やせば、いくらでも（メモリの許すかぎり）桁数の多い数を扱うことができます。ただし、桁数が多くなるほど計算には時間がかかるので、不必要に桁数を増やすべきではありません。

List 8-1 多倍長演算

```
#include <stdio.h>

// 桁数
#define N 15

// 初期化(多倍長=整数)
void init(char a[], int x) {
    int t=x;
    for (int i=N-1; i>=0; i--) {
        a[i]=t%10;
        t/=10;
    }
}

// 加算(多倍長+整数)
void add(char a[], int x) {
    int t=x;
    for (int i=N-1; i>=0; i--) {
        t+=a[i];
        a[i]=t%10;
        t/=10;
    }
}

// 加算(多倍長+多倍長)
void add(char a[], char b[]) {
    int t=0;
    for (int i=N-1; i>=0; i--) {
        t+=a[i]+b[i];
        a[i]=t%10;
        t/=10;
    }
}

// 乗算(多倍長×整数)
void mul(char a[], int x) {
    int t=0;
    for (int i=N-1; i>=0; i--) {
```

```
        t+=x*a[i];
        a[i]=t%10;
        t/=10;
    }
}

// 表示
void print(char a[]) {
    int i;
    for (i=0; i<N-1 && a[i]==0; i++);
    for (; i<N; i++) printf("%d", a[i]);
}

// メインルーチン
void main(int, char*[]) {
    int x;

    // 多倍長数を表す配列
    char a[N], b[N];

    // 多倍長+整数
    init(a, 87654321);
    printf("a: "); print(a); printf("¥n");
    add(a, x=12345679);
    printf("a+=%d: ", x); print(a); printf("¥n");
    printf("¥n");

    // 多倍長+多倍長
    init(a, 123456789);
    init(b, 987654321);
    printf("a: "); print(a); printf("¥n");
    printf("b: "); print(b); printf("¥n");
    add(a, b);
    printf("a+=b: "); print(a); printf("¥n");
    printf("¥n");

    // 多倍長×整数
    init(a, 987654321);
    printf("a: "); print(a); printf("¥n");
    mul(a, x=1000000);
    printf("a*=%d: ", x); print(a); printf("¥n");
}
```

スコアに関するデザインの指針

プログラミングテクニックとは少し違いますが、ここではスコアに関するゲームデザイン上の留意点を紹介します。

スコアの再現性

スコアにはある程度の再現性があるべきです。まったく同じようにプレイしているはずなのにプレイするたびにスコアが違ってしまうのでは、非常に博打性の高いゲームになってしまいます。それはそれで楽しいかもしれませんが、スコアが実力よりも運で決まってしまうのでは、腕を磨く動機が弱まってしまいます。

注意すべきなのは乱数の使い方です。敵や弾などを動かす際にはある程度のランダム性を入れたほうが面白いのですが、たとえばプレイするたびに出現する敵や弾の数がまったく違ってしまうようでは、スコアが運に左右されてしまいます。乱数は使いすぎず、たとえば「弾を撃つ方向」や「撃ち返し弾を撃つかどうか」といった微妙な変化をつけるためにかぎって使うとよいでしょう。

残機ボーナス、残ボムボーナス

残機や残ボム（残りボム）の数に応じてスコアにボーナスを加えると、プレイをよりよいものにする動機が強まります。ゲームによってはゲームクリア時にボーナスが入るだけではなく、ステージ単位でボーナスが入るものや、残機数や残ボム数に応じてボーナスの倍率が常に変動するようなものもあります。

永久パターンの防止

永久パターン（略して「永パ」などとも呼ばれる）というのは、ある一連のプレイを繰り返すことによって、無限に得点が稼げるパターンのことです。永久パターンには次のような場合があります。

①ボスキャラが撃ってくるミサイルなどを破壊すると得点が入り、かつボス本体を破壊せずにいつまでも粘り続けることができる

②戻り復活のゲームにおいて、あるステージで一定以上の得点を稼ぐと自機を必ず増やすことができるために、ステージの最後で故意に死んでステージの最初に戻り、えんえんと同じステージで得点が稼げる

永久パターンが発見されてしまうと、そのゲームを普通にプレイしてスコアを稼ぐ意味が失われてしまいます。そこで多くのゲームは、永久パターンを防止するためのなんらかの仕組みを取り入れています。

たとえば、①のような永久パターンを防止するには次のような方法があります。

- **一定時間内に倒さないとボスキャラの攻撃が極端に厳しくなり、いつかは必ずミスするようにする**
- **制限時間内に倒さないと、ボスキャラが逃げるようにする**
- **ボスキャラが撃ってくるミサイルなどでは得点が入らないようにする**

②のような永久パターンを防止するには次のようにします。

- **1ステージで稼げる得点の最大値を見積もっておき、自機を増やすために必要な得点をそれよりも高く設定する**
- **戻り復活ではなく、その場復活にする**

特にボスキャラと戻り復活（→ P. 164）に関しては永久パターンが発生しがちなので、ゲームをデザインする際には注意が必要です。

コンボボーナス

敵を連続して倒したり、特定の順番で倒したりすると入るボーナスのことです。コンボボーナスのルールはゲームによってさまざまですが、たとえば次のようなルールがあります。

- **敵を倒すとゲージが溜まり、そのゲージが0になる前に素早く次の敵を倒していくとボーナスが入る（「怒首領蜂（→ P. 331）」）**
- **同じ色の敵を3機ずつの組で倒していくとボーナスが入る（「斑鳩（→ P. 324）」）**

このほか、特定の種類の敵を逃さず連続で倒したり、特定のアイテムを連続して拾ったりといったボーナスが多くのゲームに見られます。

こういったコンボボーナスはゲームのクリアとは関係ないことがほとんどです。ボーナスとクリア条件を無関係にしておけば、初心者はとりあえずボーナスを気にしなくてもゲームをクリアすることができ、ゲームに慣れた上級者はボーナスを狙いながらプレイできるので、ゲームの間口が広がります。

コンティニューのペナルティ

コンティニューがあるゲームでは、コンティニューを使った場合と使わなかった場合とで、スコア表示を変えるのが普通です。多くのゲームで採用されているのは、コンティニュー回数をスコアの下1桁に入れるという方法です。たとえば、未コンティニューで100,000,000点を出した場合のスコアは、

100,000,000

ですが、1回コンティニューをした場合には、

100,000,001

になります。こうしておけばスコアランキングなどを見たときに、そのスコアがノーコンティ

ニューで出されたものなのか、コンティニューを使ったものなのかがすぐにわかります。なお、スコアの下1桁をコンティニュー表示に使うときには、敵などを倒して得るスコアは10点単位に切り上げておき、スコアの下1桁が変化しないようにしておきます。

スコアランキング

上位のスコアをランキング表として表示する機能です。プレイヤーにとってスコアランキングは、上達の目安にも技術を磨くうえでの目標にもなります。ゲーム雑誌などでは全国のゲームセンターと提携して、全国のスコアランキング表を載せていることもあります（こういった雑誌は減ってしまいましたが）。最近ではインターネットを使ったインターネットランキングも一般的になりました。

スコアランキングに関しては、ゲームの難易度設定や残機設定に注意する必要があります。難易度や残機の設定を変えられるゲームでは、そういった設定がスコアに影響しないようにするか、スコアランキングの際にきちんと設定が表示されるようにしなければなりません。

また、特にインターネットランキングなどでは不正なスコア登録への対策も必要です。対策としては、スコアを登録する際に暗号化したり、上位スコアの場合にはリプレイもいっしょに登録したりといった方法があります。

リプレイ

プレイヤーがゲームをプレイした様子をビデオのように再生する機能のことです。リプレイ機能があると、上手にプレイできた映像を保存したり、リプレイを研究してさらなる上達を目指したりといったように、ゲームの遊び方が広がります。

リプレイ機能を実現するには、ゲーム中にプレイヤーのスティック入力やボタン入力を保存しておきます（Fig. 8-6）。そしてリプレイの際には、プレイヤーによる入力のかわりに保存しておいた入力データを使ってゲームを進行させます。敵や弾などの動作はプログラムが決めるので、プレイヤーの入力が同じならば、ゲーム全体としても同じ進行が再現されるというわけです。

注意が必要なのは乱数を使う場合です。プレイ時とリプレイ時とで乱数の値が違うと、ゲームの進行が変わってしまいます。これを避けるには、プレイヤーの入力と合わせて、乱数の「種（たね）」も保存しておきます。一般にプログラムで使う乱数は、完全にランダムな値というわけではなく、種と呼ばれる数値から計算によって求められる「疑似乱数」です。したがって、乱数の種を保存しておき、ゲーム中とリプレイ中とで同じ方法を使って疑似乱数を計算すれば、どちらも同じ乱数列が得られます。計算した乱数列を全部保存しておく必要はありません。乱数に関する詳細は「乱数」（→ P. 302）で解説します。

リプレイはプレイヤーのプレイを保存する用途のほか、デモ画面などで使うこともできます。アーケードゲームの多くにはゲームの作者によるリプレイがあらかじめ入っていて、デモ画面ではこのリプレイが流れます。

List 8-2はリプレイに関してまとめたプログラムです。スティック入力やボタン入力を保存するには、各方向や各ボタンの入力状態を1ビットで表し、複数のビットをまとめてchar型などの値にします（Fig. 8-7）。char型を利用するほうが各入力状態をbool型などで保存するよりも、リプレイデータを小さくすることができます。

Fig. 8-6 リプレイ

時間

リプレイデータ

0	1	2	3	4	5	6
入力なし	左	左上	上	上	右上+ボタン1	右上

通常のプレイ時には入力を保存する

リプレイ時にはリプレイデータを入力として使う

通常プレイ

リプレイ

Fig. 8-7 入力状態の保存方法

ビット番号

各方向や各ボタンの入力をビットで表し、全部で8ビット(char型)などにまとめる

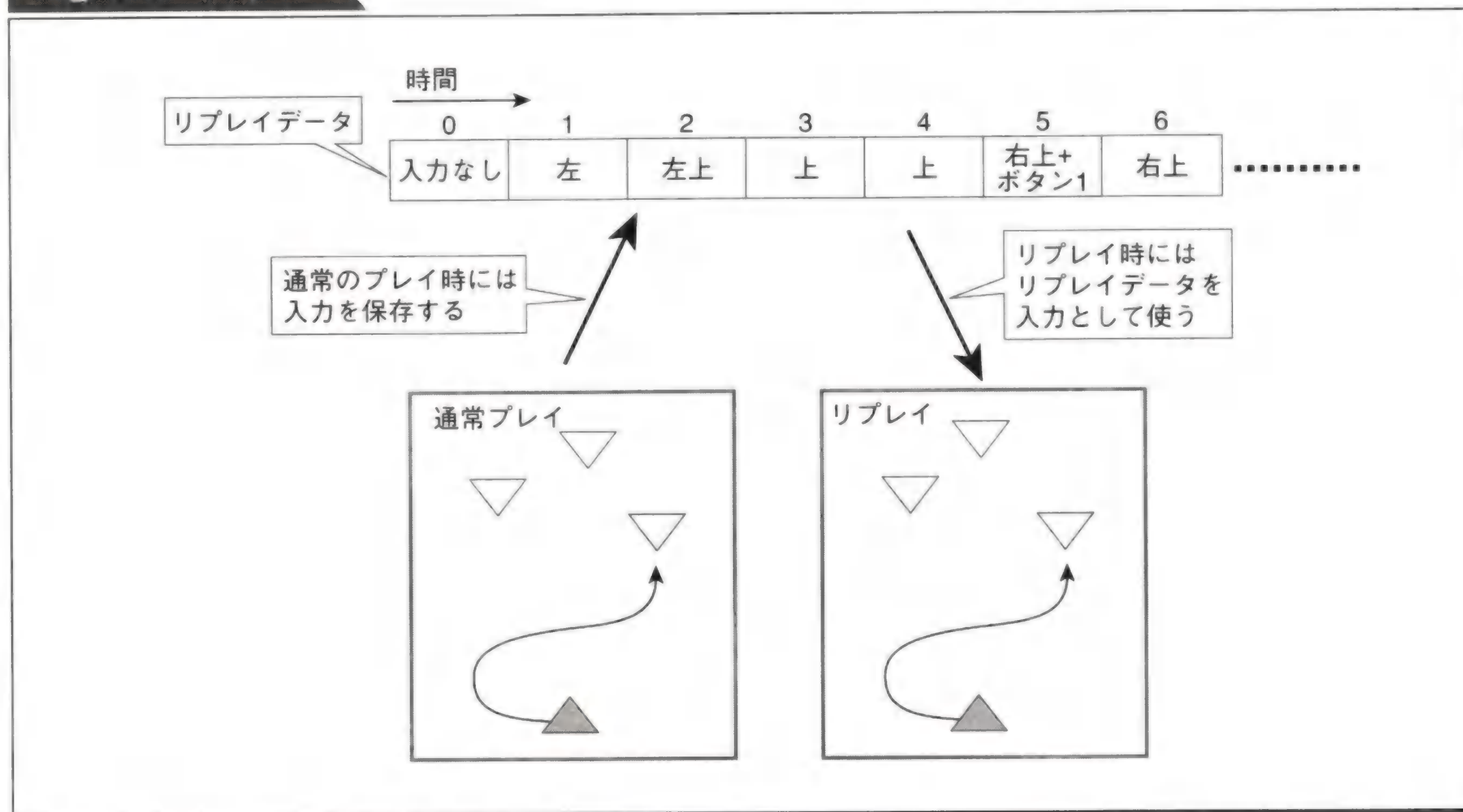
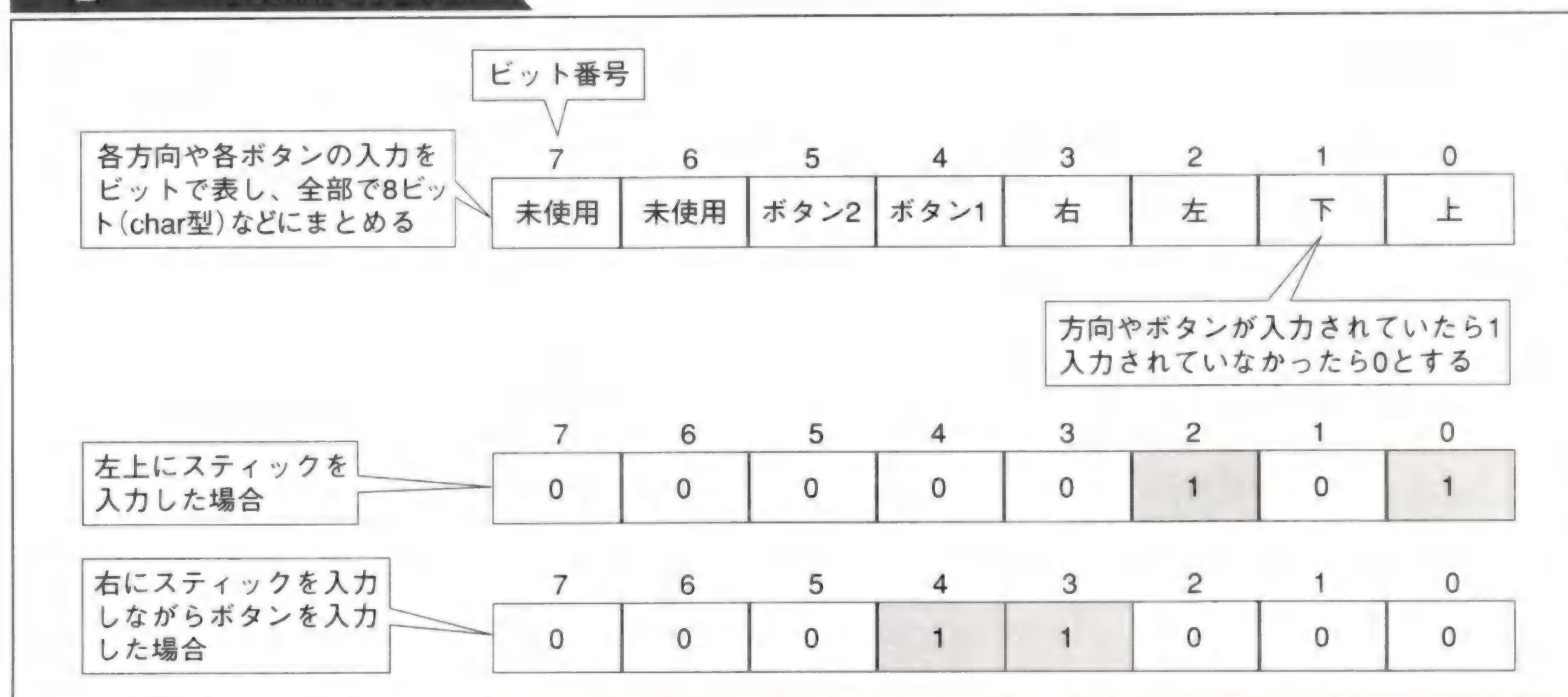
7	6	5	4	3	2	1	0
未使用	未使用	ボタン2	ボタン1	右	左	下	上

方向やボタンが入力されていたら1
入力されていなかったら0とする

左上にスティックを入力した場合

7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1

右にスティックを入力しながらボタンを入力した場合

7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0

List 8-2 リプレイ

```
// スティックとボタンの入力を表す定数:
// 各入力を1ビットで表す。
enum {
    UP=1, DOWN=2, LEFT=4, RIGHT=8,
    BUTTON1=16, BUTTON2=32
} INPUT;

// リプレイデータを保存する構造体:
// スティックとボタンの入力を保存する。
#define MAX_TIME 10000
typedef struct {
    char Input[MAX_TIME];
} REPLAY_DATA;

// 自機を動かす
void MoveMyShip(
    char input,                 // スティックとボタンの入力
    bool is_replay,             // リプレイ中ならばtrue
    REPLAY_DATA* replay_data,   // リプレイデータ
    int& time                   // タイマー
) {
    // リプレイの処理:
    // リプレイ中ならば入力をリプレイデータから読み出す。
    // リプレイ中でなければリプレイデータに入力を保存する。
    if (is_replay) {
        input=replay_data->Input[time];
    } else {
        replay_data->Input[time]=input;
    }

    // タイマーを進める
    time++;

    // 入力に従って自機を動かす:
    // 移動や射撃の具体的な処理は、
    // MoveUp、MoveDown、MoveLeft、MoveRight、
    // Shot、Laserの各関数で行うとする。
    if (input&UP    ) MoveUp();
    if (input&DOWN  ) MoveDown();
    if (input&LEFT  ) MoveLeft();
    if (input&RIGHT) MoveRight();
    if (input&BUTTON1) Shot();
    if (input&BUTTON2) Laser();
}
```

乱数

乱数（乱数列）というのは数が不規則に並んだ数列のことです。シューティングゲームでは主に、敵や弾などの動きにランダム性（不規則性）を加えたいときに乱数を使います。

プログラムで使うほとんどの乱数は、完全に不規則な「本当の乱数」ではなく、「種（たね）」と呼ばれる数値から計算によって求められる「疑似乱数」です。種に対してなんらかの計算を加えることによって、1つの乱数を得ます。次はその乱数を新たな種にして、その次の乱数を計算します。このようにして、次々と乱数を得ることができます。

乱数の種

C/C++で乱数を得るのにはrand関数を使います。rand関数を使う前には、srand関数を使って乱数の種を設定する必要があります。種には好きな値を設定できますが、同じ種を設定するとまったく同じ乱数列が生成されることに注意が必要です。逆にこのことを利用して、リプレイ（→ P. 299）時にプレイ時と同じ種を設定すれば、プレイ時と同じ乱数列を得ることができます。

乱数の種はゲームの起動時などに1回だけ設定すれば大丈夫です。ゲームを起動するたびに乱数列を変えたいときには、現在時刻をミリ秒単位で表した値などを種に使います。現在時刻をミリ秒単位で表した値はゲームを起動するたびにたいてい異なるので（PCの場合）、種を変化させることができるからです。

乱数の生成方法

rand関数が乱数の生成に使う計算方法は、使用するライブラリによって違う可能性があります。A社のC/C++コンパイラのrand関数とB社のC/C++コンパイラのrand関数では、同じ種を与えても、生成する乱数が違うかもしれません。また同じコンパイラでも、バージョンが違えば生成する乱数が異なるかもしれません。

シューティングゲームの場合、乱数の計算方法が変わるのはやっかいです。ゲームの開発環境を変えたり別のOSに移植したりすると、プログラムは変わっていなくても、乱数が変わるためにゲームバランスが変化してしまう可能性があります。もっと深刻なのはリプレイ機能を使うときです。リプレイデータとして乱数の種を保存しておいても、乱数の計算方法が変わってしまうと、正常にプレイを再現することができません。計算方法が違うと、種は同じでも生成する乱数列が違ってしまうからです。

こういった問題を避けるには、ライブラリに用意されているrand関数やsrand関数は使わず、自前でrand関数やsrand関数に相当する関数を用意します。乱数の生成方法にはいろいろありますが、もっともシンプルなのは「線形合同法」という方法です。乱数の種をseedとすると、線形合同法による乱数は次のような式で簡単に得られます。

```
seed*a+b
```

bには1などの奇数を使い、aには次のような値を使います。

69069, 1664525, 39894229, 48828125, 1566083941, 1812433253, 2100005341

なお、これらの値の選び方や線形合同法以外の乱数生成方法に関しては『C言語によるアルゴリズム事典』(奥村晴彦著、技術評論社、ISBN4-87408-414-1) などが参考になります。

上記の式で乱数を生成したら、その乱数を新たな種として、次の乱数を生成するために使います。List 8-3はこの処理をまとめたプログラムです。このように自前で乱数生成処理を作っておけば、開発環境が変わった場合にも乱数が変わる心配がありません。

List 8-3 乱数の生成

```
// 乱数の種
typedef unsigned int ulong;
static ulong seed=1;

// 種の初期化
void srand(ulong s) {
    seed=s;
}

// 乱数の生成
ulong rand() {
    seed=seed*48828125UL+1;
    return seed;
}
```

難易度

難易度とはゲームの難しさのことです。最近のゲームの多くは、プレイヤーのレベルに応じてプレイ中に刻々と難易度が変化します。基本的にはプレイヤーがうまくプレイするほど難易度が上がるようになっています。具体的には次のような難易度の変化があります。

周回による難易度の上昇

全ステージクリアを1周としたときに、2周目以降があるゲームに見られる難易度の変化です。同じステージでも1周目と2周目では難易度がまったく違います。

時間による難易度の上昇

プレイ時間が長くなるほど難易度が上昇するものです。普通にプレイしていてもプレイ時間

が経過して難易度は上昇しますが、特にボスなどで点を稼いでいるとプレイ時間が長くなり、難易度はより速く上昇します。

得点による難易度の上昇

プレイヤーが得点を稼ぐほど難易度が上昇するものです。得点を稼ぐプレイヤーはうまいプレイヤーだと判断して、難易度を上げます。

装備による難易度の上昇

自機がパワーアップするゲームで、自機の装備によって難易度が変化するものです。たとえばレーザーやバリアをつけると難易度が上がって敵の攻撃が激しくなる、といったケースがあります。また、所持しているボムの数で難易度が変化することもあります。

ミスによる難易度の下降

ミスで自機を失うと難易度が下がるというものです。残機数によって難易度が変化するゲームもあります。「バトルガレッガ(→ P. 332)」などは「難易度を下げるために故意にミスをして自機を減らす」という独特のプレイスタイルで有名なゲームです。

ランク設定による難易度の変化

難易度選択機能があるゲームでは、選んだランク(イージー、ノーマル、ハードなど)によって難易度が変わります。難易度選択機能をつけるときには、たとえばハードではスコアが多く稼げるなど、難しいランクになんらかの利点を設けてもよいでしょう。

※

難易度が変わったときのゲーム内容の変化としては次のような例があります。多くのゲームは、以下のうちのいくつかを組み合わせて難易度を調整しています。

敵や弾のスピード

難易度が上がると敵や弾のスピードが上がります。これは敵や弾のスピードを変数で持っておき、難易度に応じて値を大きくすれば実現できます。たとえば、難易度1に比べて難易度2ではスピードを10%アップさせる、といった要領です。

敵が弾を撃つ数

難易度が上がると敵が撃つ弾の数が多くなります。たとえば、難易度1では1発撃つところを難易度2では2発撃つといった要領です。

弾数を細かく調整したいときには、弾を撃つ頻度を変える方法もあります。たとえば、難易度1では敵が0.5秒ごとに1発弾を撃つところを、難易度2では0.4秒ごとに1発撃つようにすれば、弾の数は増えます。

撃ち返しの確率

敵を破壊したときに自機に向かって狙い撃ち弾を発射するのが「撃ち返し」です(→ P. 257)。

撃ち返しが多いと難易度は上がります。たとえば、難易度1では撃ち返し率を0%、難易度2では撃ち返し率を20%などとして、乱数を使って撃ち返すかどうかを決めます。撃ち返しの方向に多少のランダム性を混ぜても面白くなります。

敵の耐久力

難易度が上がると敵の耐久力が上がって、敵を破壊しにくくなります。たとえば、難易度1では耐久度を100、難易度2では耐久度を120などとします。耐久力は敵の種類によって異なるので、敵ごとに難易度別の耐久力を設定しておく必要があります。

敵の出現数

難易度が上がると敵の出現数が増えたり、難易度が低いときには出なかった敵が出るようになったりします。これは敵ごとに、難易度に応じた出現数や、出現する確率などのデータを用意しておけば実現できます。

※

以上をまとめると、難易度に関する処理の基本は、難易度に応じたテーブル（データ）を用意しておくことです（Fig. 8-8）。敵や弾を動かすときには、このテーブルを参照して出現確率や耐久力を決めます。

Fig. 8-8 難易度に応じたテーブルの例

難易度	敵のスピード（倍率）	弾のスピード（倍率）	撃ち返しの確率	敵の耐久力（倍率）	敵1の出現数
1	1.0	1.0	0.0	1.0	5
2	1.0	1.1	0.2	1.0	5
3	1.1	1.2	0.3	1.3	5
4	1.1	1.3	0.4	1.3	6
5	1.2	1.4	0.6	1.6	6

…

⋮

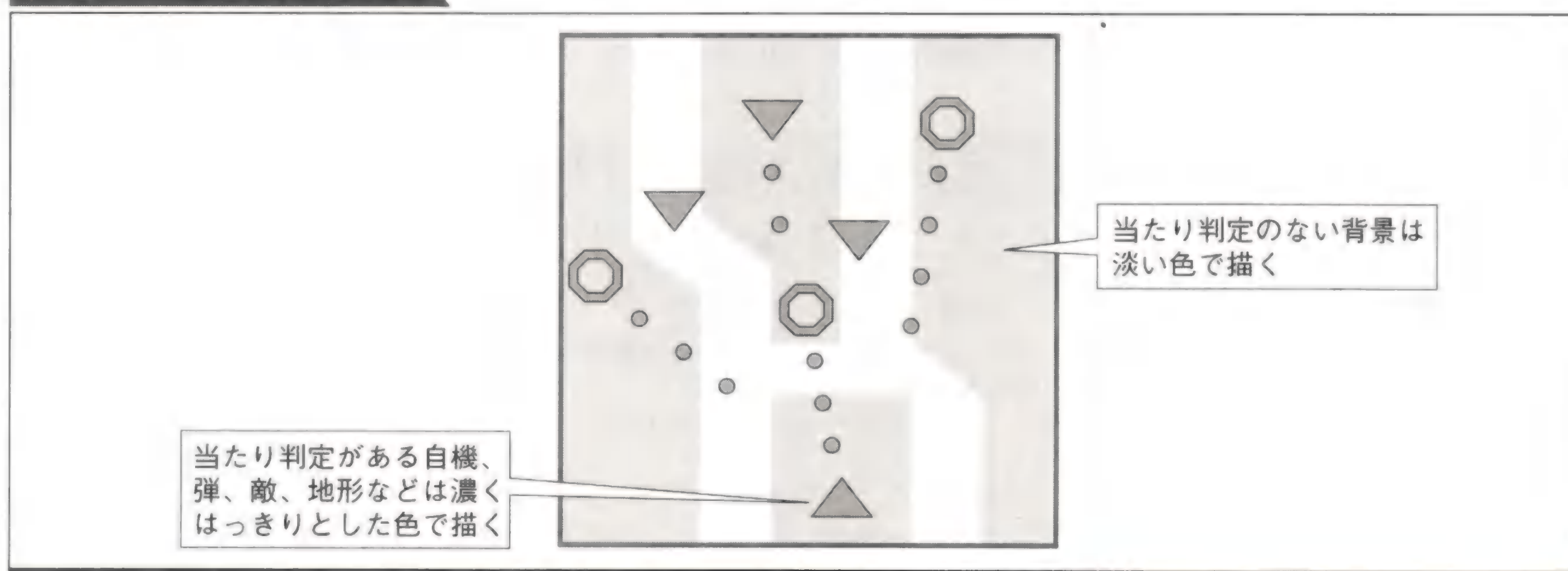
ゲームを快適にするために

ゲームは本来面白くあるべきものですが、残念ながら世の中にあるすべてのゲームが痛快というわけではありません。絵が下手だとか音楽が貧弱だとか、ゲームがつまらないのにはさまざまな要因がありますが、シューティングゲームの場合にはプログラマの努力しだいでゲームをずっと面白くすることができます。ここでは、快適に遊べるシューティングゲームをデザインする際の「気配り」のヒントを紹介します。

見やすい配色

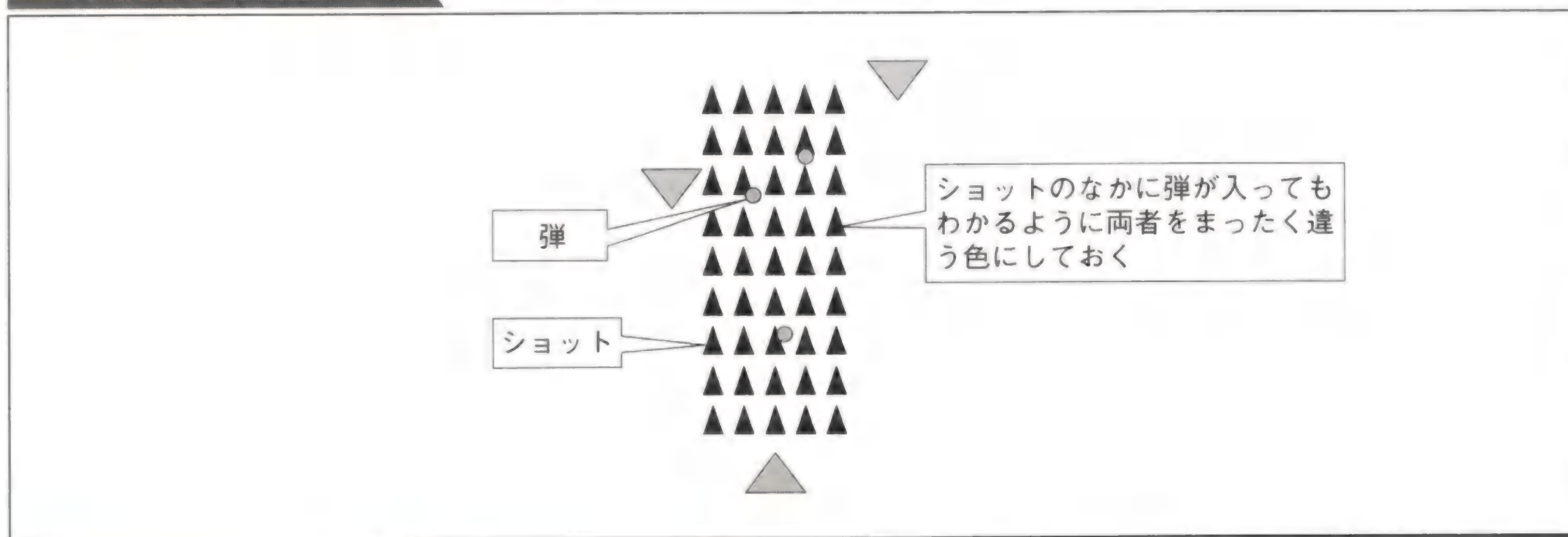
画面が見づらいゲームは遊びにくいものです。特にシューティングゲームは自機や弾などの視認性が命なので、見やすさを重視して配色する必要があります。

配色の基本方針は「当たり判定のある自機、弾、敵、地形などは濃くハッキリと描く」ことと、「当たり判定のない背景などは淡く描く」ことです（Fig. 8-9）。背景がギラギラしすぎているゲームは、背景のなかに自機や弾がとけ込んでしまいますし、長時間遊んでいると疲れます。

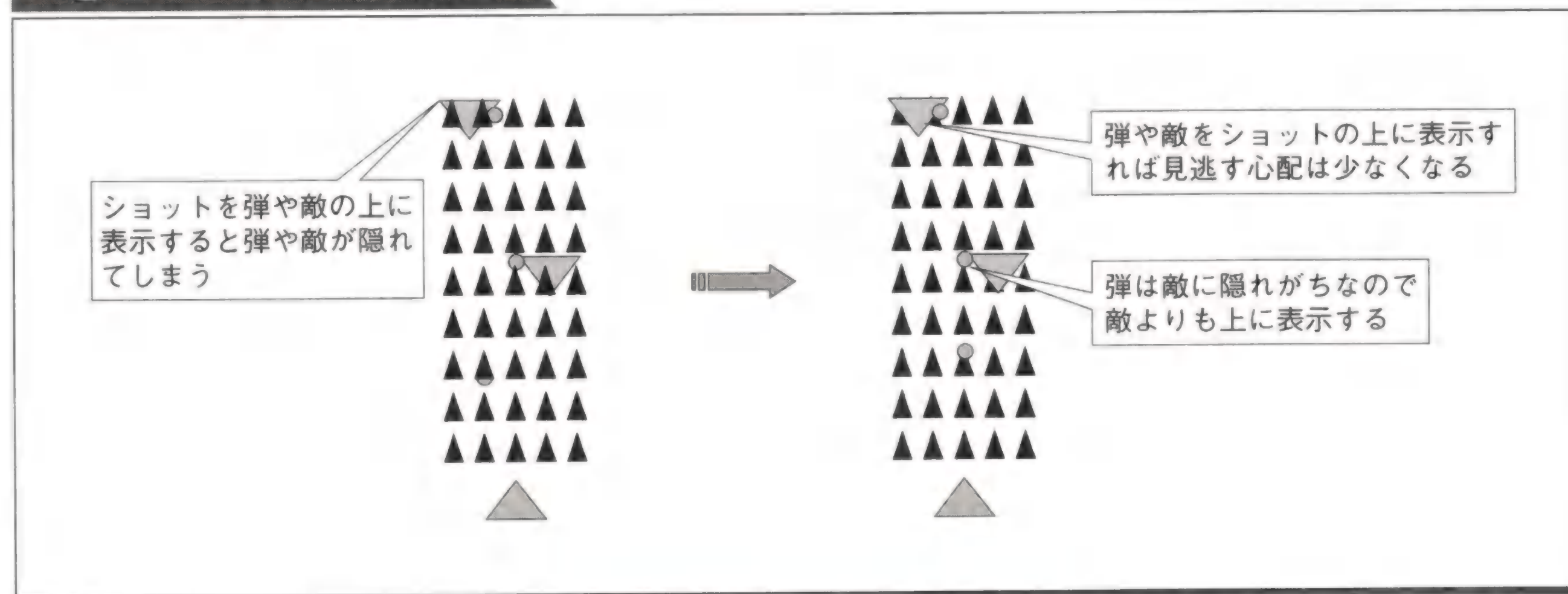
Fig. 8-9 見やすい配色

見やすい弾

配色に関連しますが、「自機の撃つショット」と「敵の撃つ弾」の色を変えることも重要です。ショットと弾は重なることが多いので、弾がショットのなかにとけ込んでしまうことがあります。たとえば「ショットは青」、「弾は赤」といったようにまったく違う色にしておけば、弾を見落としにくくなります（Fig. 8-10）。

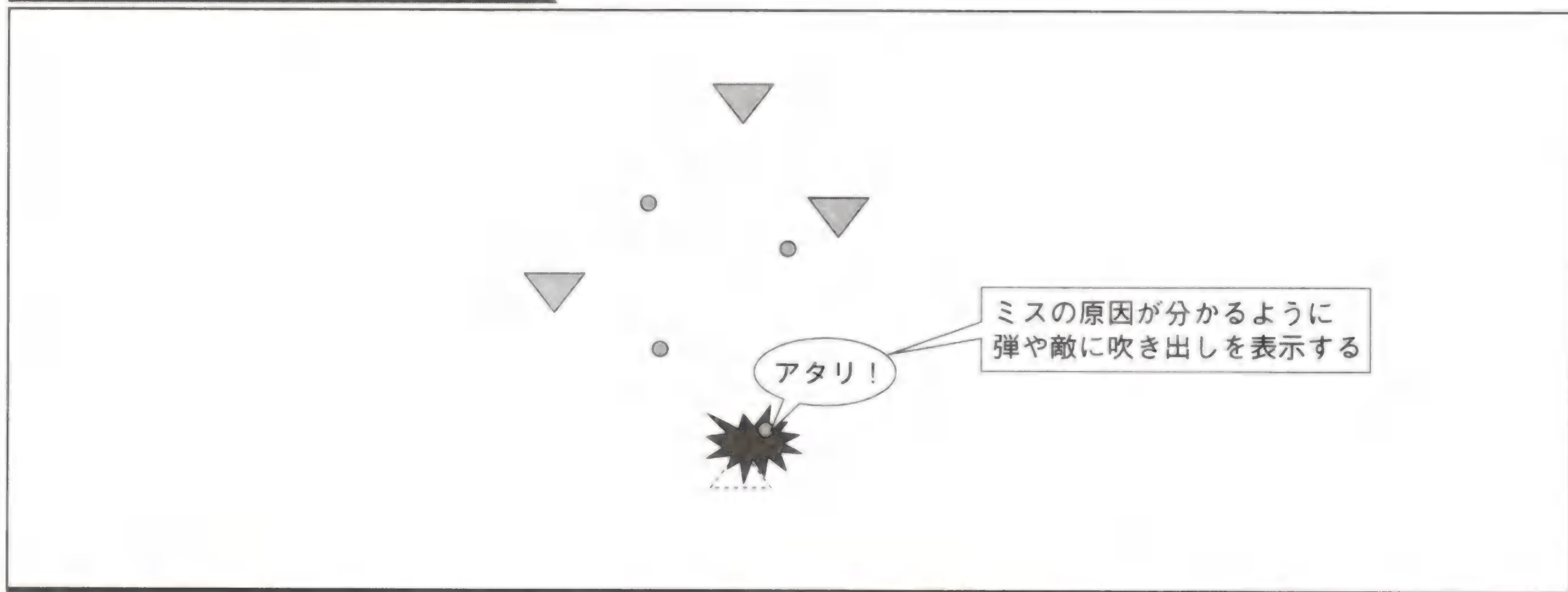
ゲームによっては弾の配色を厳格に決めているものもあります。たとえば「弾は必ずオレンジ」といった具合です。一方で、何色ものカラフルな弾が出てくるゲームもあります。前者はデザインの幅が制限されますが、弾は見やすくなります。後者はデザインの幅が広がりますが、視認性が悪くならないように気をつかう必要があります。

Fig. 8-10 見やすい弾

見やすい重ね合わせ

ショット、弾、敵などを表示する順序も重要です。たとえばショットが弾や敵の上に表示されてしまうと、弾や敵が見えなくなってしまいます。ショットは一時的に見えなくなっても大丈夫ですが、弾や敵を見逃すのは致命的です。そこで、より重要なものを上に表示するようにします。たとえば、弾を一番上、次に敵、最後にショットという順に表示したほうが遊びやすくなります(Fig. 8-11)。

Fig. 8-11 見やすい重ね合わせ

やられた原因の表示

これは「セクシーパロディウス(→ P. 329)」などにある面白い仕組みで、自機が接触した弾や敵などに吹き出しが出て、ミスの原因を表示してくれるというものです(Fig. 8-12)。あまり多くのゲームには採用されていませんが、この仕組みを入れておけば「いまなんでやられたの!?」と理不尽に感じる場面が少なくなるかもしれません。

Fig. 8-12 やられた原因の表示

複数人数プレイ

複数人数でゲームを遊ぶのは楽しいものです。シューティングゲームの場合には複数人数だと弾筋が乱れて遊びにくくなるケースもありますが、可能ならば複数人数プレイに対応したほうが遊びの幅は広がります。複数人数プレイの場合には、スコアや難易度、出現アイテム数などを調整し直す必要があります。

繰り返しプレイのための動機

プレイするたびに何か嬉しいことがあると、プレイヤーはそのゲームを繰り返し遊ぼうと思うはずです。昔のゲームのように「自分の技量が向上していくことだけが嬉しい」というストイックなゲームデザインも魅力あるものですが、最近ではもう少し柔らかめに「クリアするたびになんらかのご褒美が用意してある」というデザインのほうが主流になっています。

よく見かける「ご褒美」は、クリアするたびにムービーやCGが増えるといったものです。面白いのは「R-TYPE FINAL（→ P. 323）」で、このゲームでは多数の自機が用意してあり、プレイ時間に応じて使える機体が増えていきます。これは単なるご褒美ではなく、新しい機体で繰り返しゲームを遊べるという点で、ユニークな工夫だといえるでしょう。

多彩なゲームモード

普通にゲームを遊ぶだけではなくて、特定の条件下で高いスコアを目指すスコアアタックモードや、短い時間でのクリアを目指すタイムアタックモードなどがあると、ゲームの遊び方が広がります。また、家庭用ゲームでは、特定の場面だけを繰り返し練習できるようなモードがあると、上達の助けになって便利だと感じるプレイヤーも多いでしょう。

画面や音のアレンジ

あまり市販のゲームでは見ませんが、画面や音の出力方法にアレンジが加えられると、好みに合わせた楽しみ方ができます。たとえば、画面に関しては走査線ふうの表示ができるとか、音に関してはサラウンドに対応するといったことが考えられます。

フレームレートの調整

フレームレートというのは1秒間に表示を更新する回数のことです。特にPCの場合には画面モードによって最適なフレームレートが変わるので、ゲームのフレームレートに合った画面モードを選ぶか、逆に画面モードに合わせてゲームのフレームレートを変えるかする必要があります。比較的簡単に実現できるのは前者ですが、後者の方法をとれば、動作環境に応じて最適な画面モードを選ぶことができます。

Stage 8 のまとめ ▶▶

シューティングゲームというのは、できばえがプログラマの力量に左右される度合いが強いジャンルだといえます。美しいグラフィックや派手なアニメーションを作ったとしても、それをゲームとして面白いものにしあげられるかどうかは、プログラマの地道な努力にかかっているのです。

というわけで、「隙のないシステム作りが面白いゲームを作る！」というのが本章のまとめです。

付録
Appendix

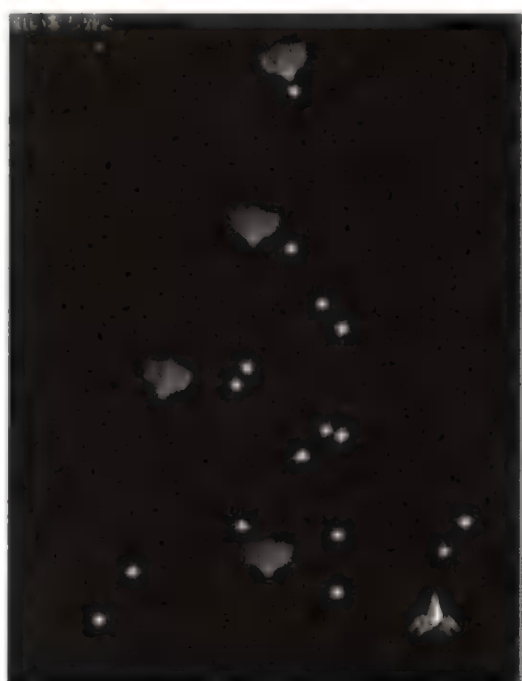
Appendix 1　デモプログラム一覧
Appendix 2　引用ゲーム一覧
Appendix 3　索引

Appendix 1 デモプログラム一覧

Stage 02 弾 *Bullet*

狙い撃ち弾
➡P. 10「狙い撃ち弾」

方向弾
➡P. 23「方向弾」

狙い撃ち弾2
➡P. 10「狙い撃ち弾」

テーブルを使った方向弾
➡P. 24「テーブルを使った方向弾」

狙い撃ち弾(DDA)
➡P. 15「DDAを使って弾を動かす」

方向弾(DDA)
➡P. 26「DDAを使った方向弾」

狙い撃ち弾(固定小数点数)
➡P. 18「固定小数点数を使って弾を動かす」

n-way弾
➡P. 33「n-way弾」

円形弾
➡P. 36「円形弾」

分裂弾
➡P. 38「分裂弾」

簡易誘導弾
➡P. 39「誘導弾」

誘導弾
➡P. 39「誘導弾」

誘導レーザー
➡P. 46「誘導レーザー」

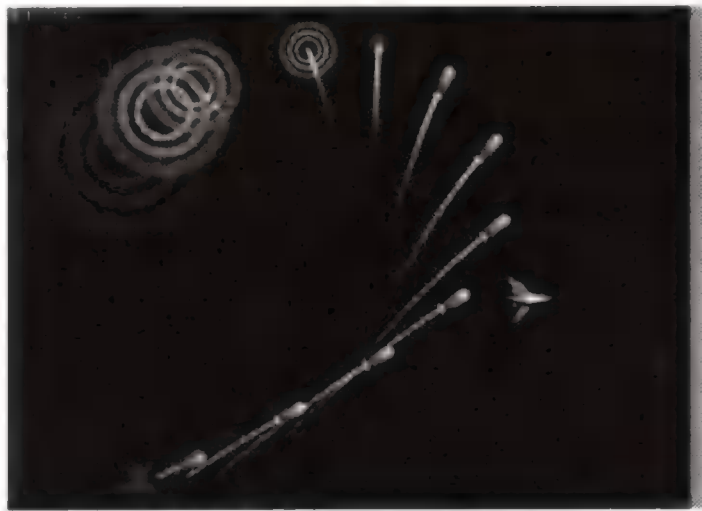
誘導レーザー2
➡P. 46「誘導レーザー」

誘導ミサイル
➡P. 52「ミサイル」

加速n-way弾
➡P. 54「加速弾」

加速誘導レーザー
➡P. 54「加速弾」

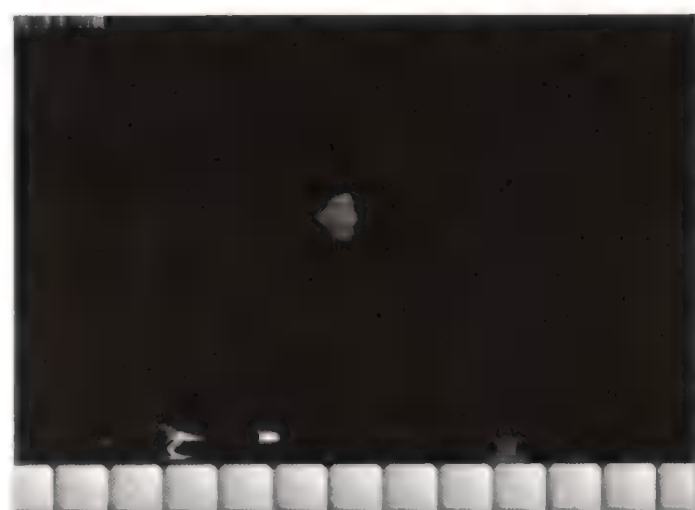
落下弾
➡P. 56「落下弾」

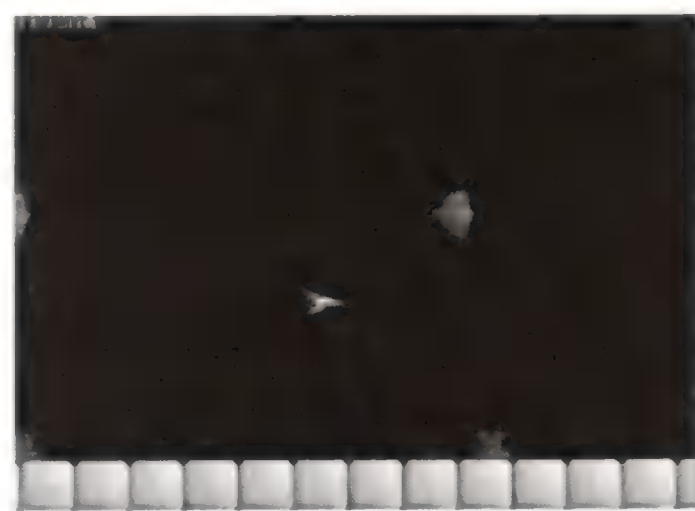
乱数
➡P. 70「弾の動きとランダム性」

乱数2
➡P. 70「弾の動きとランダム性」

Stage 03 自機 *MyShip*

ワープ移動
➡P. 85「ワープ移動」

合体する自機
➡P. 91「合体する自機」

ボタンによるスピード調整
➡P. 88「ボタンによる自機の
スピード調節」

地上を歩く自機
➡P. 93「地上を歩く自機」

アイテムによるスピード調整
➡P. 89「アイテムによる自機の
スピード調節」

変形する自機
➡P. 96「変形する自機」

水中の移動
➡P. 98「水中の移動」

バリア
➡P. 108「バリア」

オプション
➡P. 104「オプション」

Stage 04 武器 *Weapon*

基本のショット操作
➡P. 114「基本のショット操作」

溜め撃ち
➡P. 117「溜め撃ち」

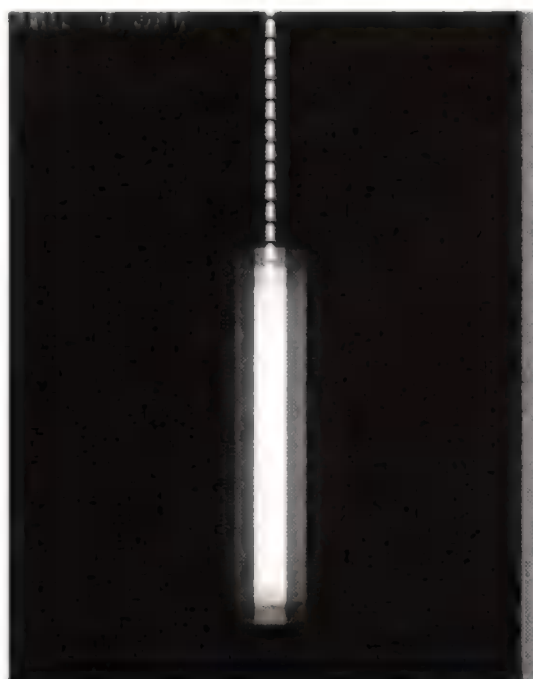
連射
➡P. 116「連射」

セミオート連射
➡P. 119「連射と溜め撃ちの共存（セミオート連射）」

Stage 4 >>

ボタンを離して溜める溜め撃ち
➡ P. 124「ボタンを離して溜める溜め撃ち」

Stage 4 >>

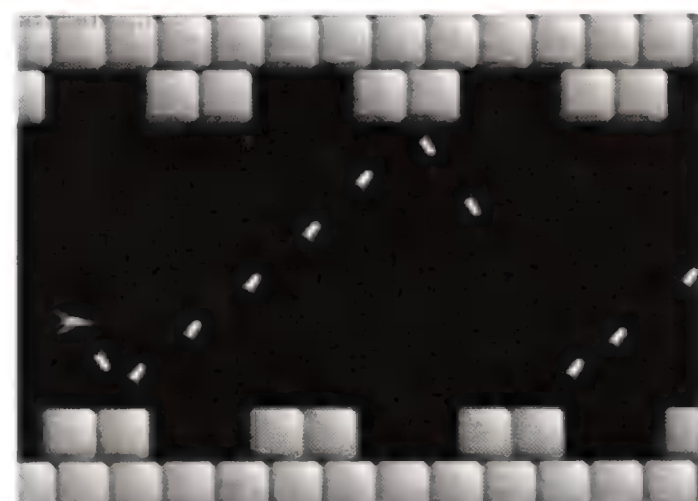
ロックオンレーザー
➡ P. 144「ロックオンレーザー」

Stage 4 >>

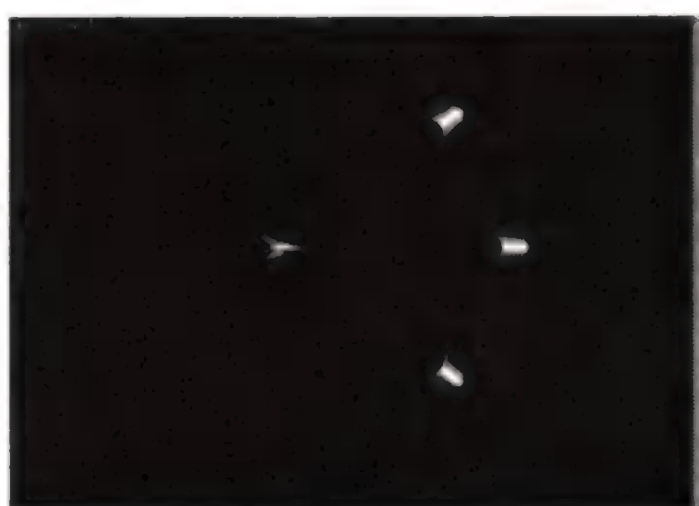
連射とレーザーの共存
➡P. 126「連射とレーザーの共存」

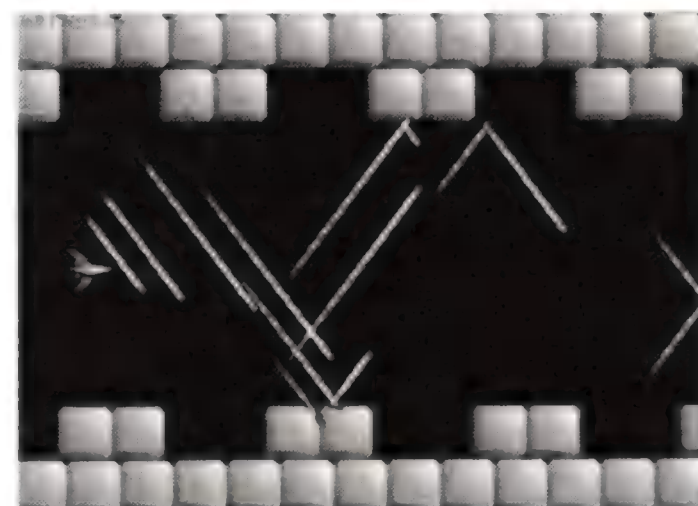
Stage 4 >>

地形に沿って飛ぶミサイル
➡P. 148「地形に沿って飛ぶミサイル」

Stage 4 >>

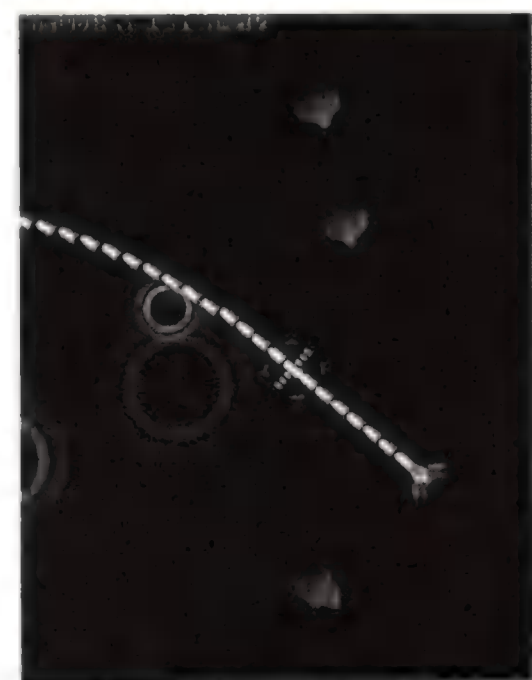
ロックショット
➡P. 129「ロックショット」

Stage 4 >>

地形で反射するショット
➡P. 151「地形で反射するショット」

Stage 4 >>

コマンドショット
➡P. 131「コマンドショット」

Stage 4 >>

地形で反射するレーザー
➡P. 153「地形で反射するレーザー」

Stage 4 >>

照準を使った爆撃
➡P. 141「照準を使った爆撃」

Stage 4 >>

方向切り替えによる全方位射撃
➡P. 157「方向切り替えによる全方位射撃」

Stage 05 特殊攻撃 *Special*

ボム
➡P. 162「ボム」

無敵状態
➡P. 172「無敵状態」

パンチ
➡P. 165「近接攻撃」

バーサーク
➡P. 174「バーサーク状態」

誘爆
➡P. 167「誘爆」

味方をつかんで投げる
➡P. 178「味方をつかんで投げる」

アイテムによる特殊攻撃
➡P. 169「アイテムによる特殊攻撃」

敵をつかんで投げる
➡P. 181「敵をつかまえて投げる」

敵をつかまえて味方にする
➡P. 185「敵をつかまえて味方にする」

自機の色を切り替えて弾を吸収する
➡P. 195「自機の色を切り替えて弾を吸収する」

敵につかまった自機を取り返してパワーアップする
➡P. 188「敵につかまった自機を取り返してパワーアップする」

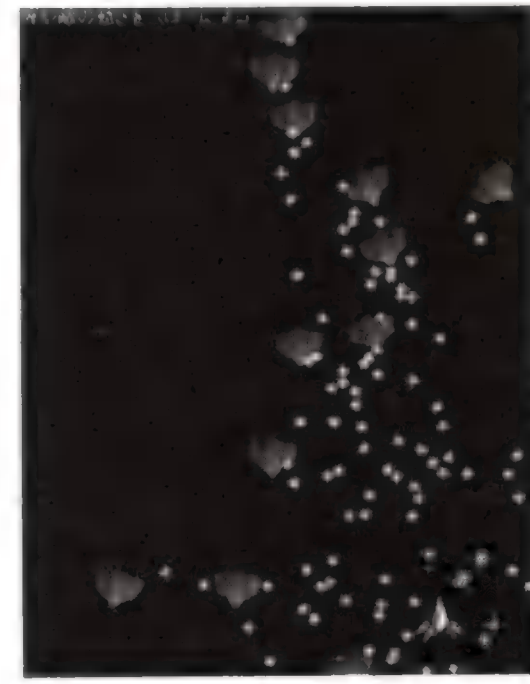
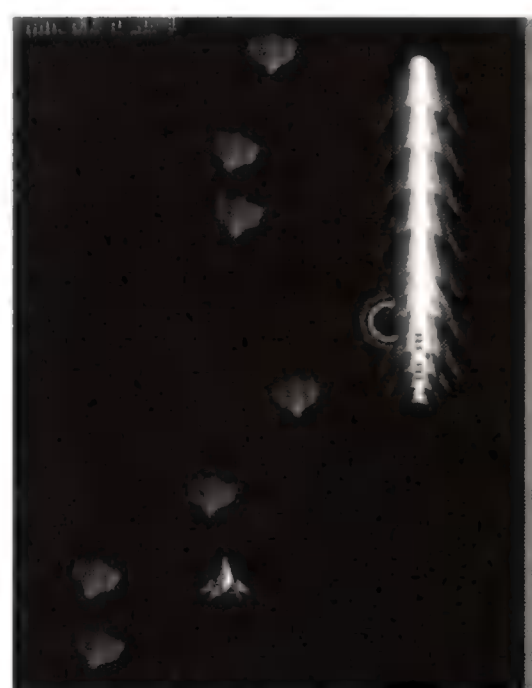
敵の弾をショットとして跳ね返す
➡P. 197「敵弾をショットとして跳ね返す」

味方に接近してショットを強化する
➡P. 190「味方に接近してショットを強化する」

レーザー同士をぶつける
➡P. 200「レーザー同士をぶつける」

味方がいる方向に強いショットを撃つ
➡P. 192「味方がいる方向に強いショットを撃つ」

敵弾に自機をかすらせる
➡P. 202「敵弾に自機をかすらせてパワーアップする」

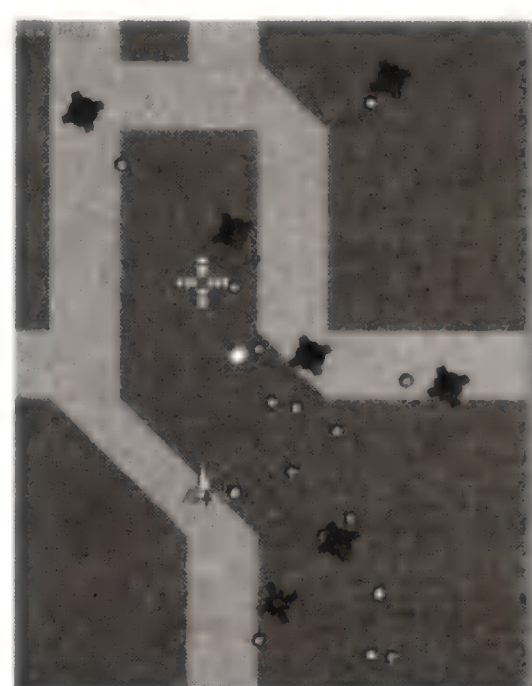
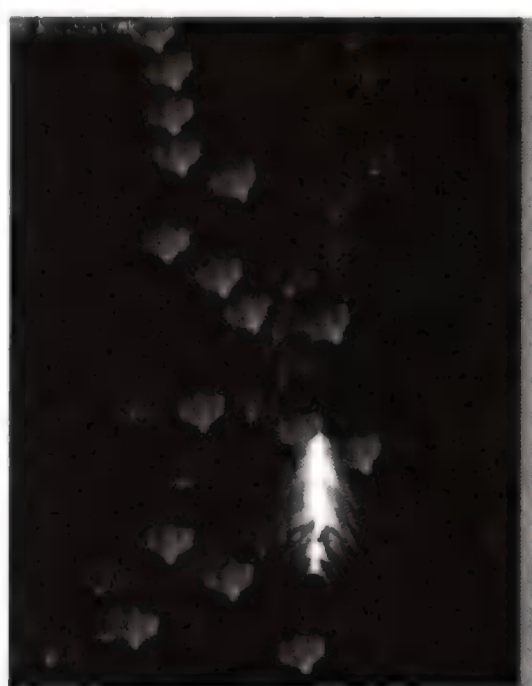
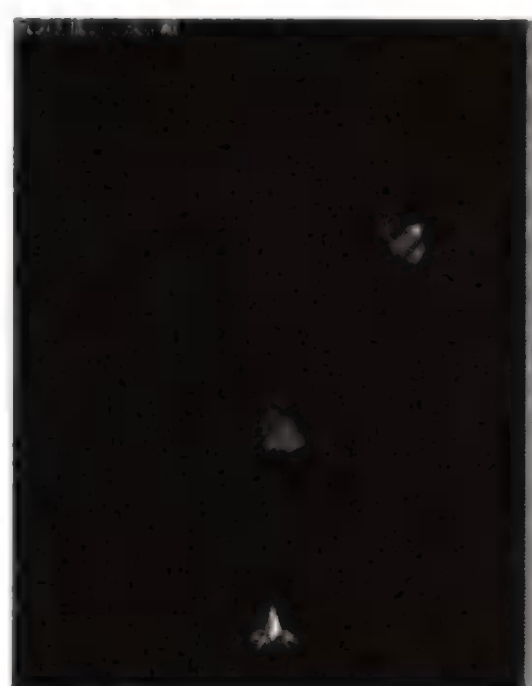
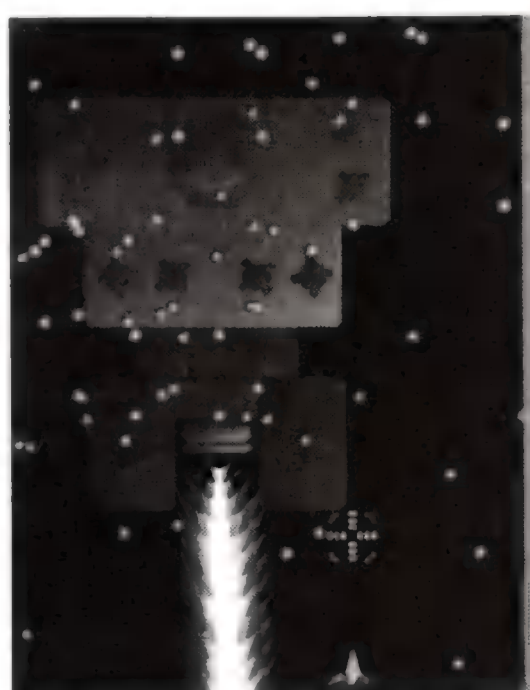
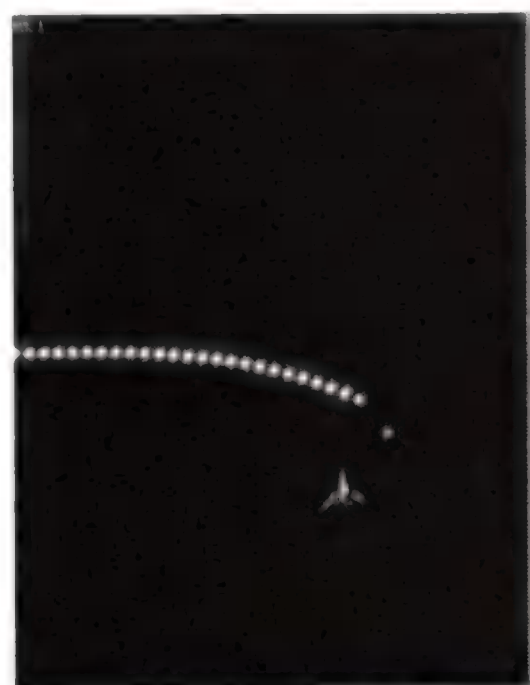
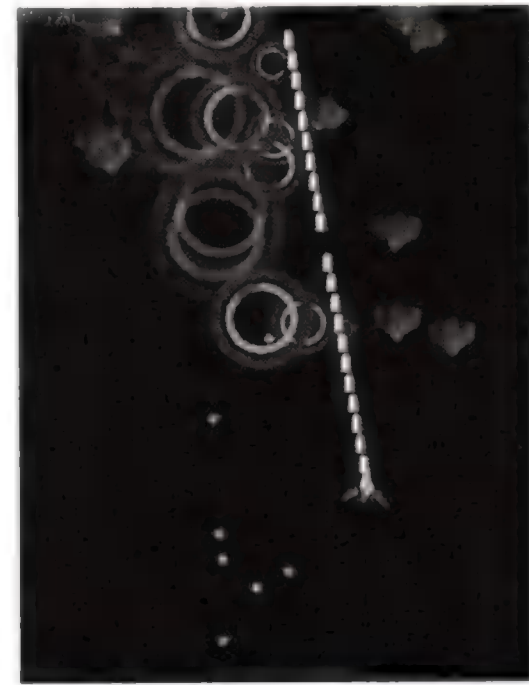
味方に当てたショットを強化する
➡P. 193「味方に当てたショットを強化する」

弾や敵の動きをスローにする
➡P. 205「弾や敵の動きをスローにする」

Stage 6 >>

ボス
➡P. 237「ボスキャラの構造」

Stage 6 >>

多関節
➡P. 250「多関節」

Stage 6 >>

触手
➡P. 246「触手」

Stage 6 >>

撃ち返し
➡P. 257「撃ち返し」

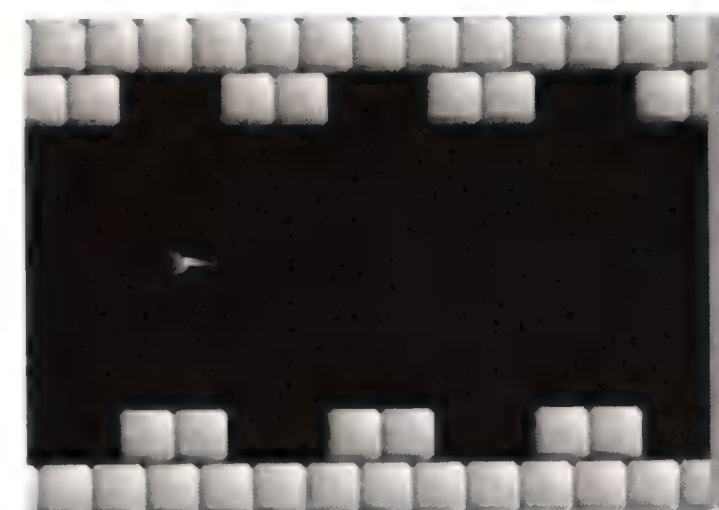
Stage 07 背景 *Scroll*

Stage 7 >>

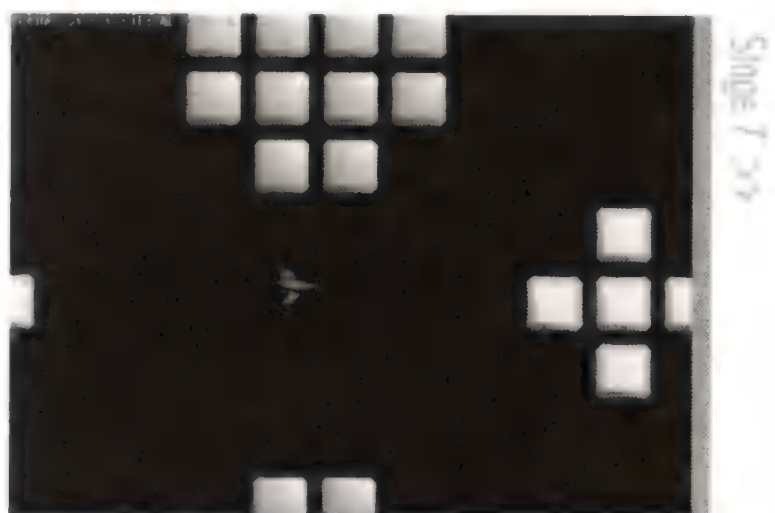
背景の表示
➡P. 270「背景の表示」

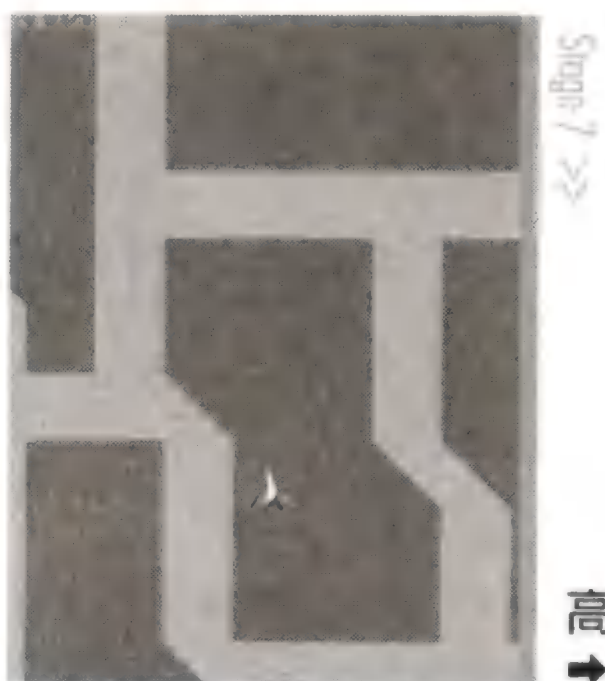
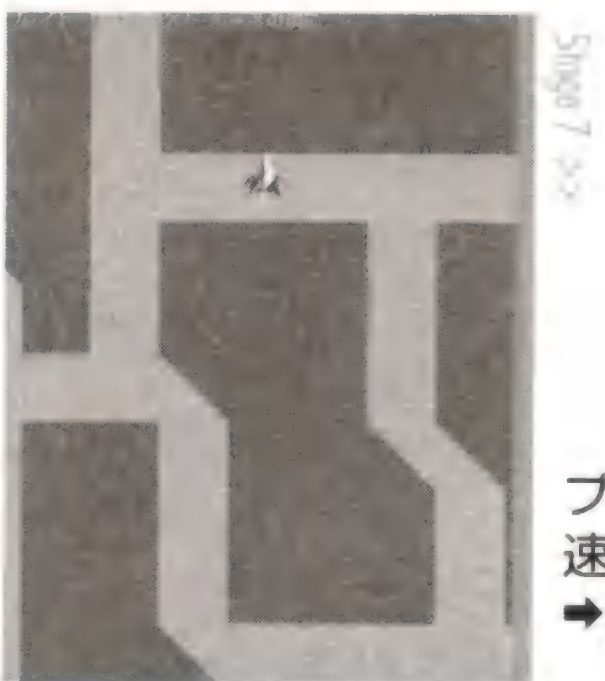
Stage 7 >>

星のスクロール
➡P. 274「星のスクロール」

Stage 7 >>

多重スクロール
➡P. 272「多重スクロール」

Stage 7 >>

強制縦スクロール+
限定横スクロール
➡P. 276「強制縦スクロール+
限定横スクロール」

強制横スクロール+任意縦スクロール
➡P. 279「強制横スクロール+任意縦スクロール」

回転
➡P. 281「回転」

高速スクロール
➡P. 283「高速スクロール」

プレイヤーによるスクロール
速度の調整
➡P. 286「プレイヤーによる
スクロール速度の調節」

Appendix 2 引用ゲーム一覧

本文中で引用したゲームの一覧です。PCや家庭用ゲーム機に移植されている作品も多いので、気になるゲームはぜひ一度、実際に遊んでみることをお勧めします。なお、一部のゲームには続編が出ていたり、移植の際にタイトルが変わっていたりする場合があるので、詳しくは各メーカーのWebサイトやショップなどでお確かめください。

※凡例

●ゲーム名

・メーカー：発売年度：プラットフォーム

AC：アーケード、DC：ドリームキャスト、GC：ゲームキューブ、
PC：PC/AT互換機（Windowsパソコン）、PS：プレイステーション、
PS2：プレイステーション2、SS：セガサターン、XB：Xbox

・スクロール方向、本書内で取り上げた主な特色

コメント

●R-TYPE

・アイレム：1987：AC/PS
・横スクロール、溜め撃ち/地形で反射するレーザー

「グラディウス」と並んで有名な横スクロールゲームのシリーズ。「波動砲」や「フォース」を使った戦略性の高い（ときにはパターン性の強い）ゲーム性と、緻密に描き込まれたグラフィックの美しさが特徴です。

●R-TYPE FINAL

・アイレム：2003：PS2
・横スクロール、プレイ時間に応じて自機の種類が増える

「R-TYPE」シリーズの最新&最終作。性能や性質が違う膨大な種類の自機を集めたり、AI制御の自機同士を対戦させたりと、独特の楽しみ方が用意されています。

●アサルト

・ナムコ：1988：AC/PS
・全方向スクロール、任意回転

システムII基板の回転機能を活用したゲーム。自機は戦車で、旋回すると画面全体が回転します。2Dグラフィック全盛の当時、回転はずいぶん派手な演出に見えたものです。

●アフターバーナー

・セガ：1987：AC/DC/PS2/SS
・疑似3D

疑似3Dのドッグファイトゲーム。戦闘機（F14トムキャット）を操って、画面奥から迫る敵機を自動追尾ミサイルとバルカン砲で撃墜します。セガの体感シリーズの1つで、大きなコクピット型筐体に乗り込んで遊びました。

●斑鳩（いかるが）

・トレジャー：2001：AC/DC/GC
・縦スクロール、自機の属性が黒と白に変化、コンボボーナス

自機の属性を切り替えて弾を吸収する、という独特のルールを採用したゲーム。白と黒を基調とした美しいグラフィックスと、迫力あるオーケストラサウンド、戦略性（とパターン性）の強いゲーム性が特徴です。

●SDI

・セガ：1987：AC
・固定画面、トラックボールで照準を操作

ボタンつきのジョイスティックとトラックボールを使って、自機と照準を同時に操作するユニークなゲーム。上手な人のプレイは、まるで両手が別々に思考しているかのようで感動的です。

●エスプレイド

・アトラス/ケイブ：1998：AC
・縦スクロール、バリア

超能力者が主人公のゲーム。ケイブらしい分厚い弾幕が楽しめます。コンシューマ機への移植は残念ながらありませんが、携帯アプリとしては移植されています。

●エスプガルーダ

・ケイブ：2003：AC/PS2
・縦スクロール、バリア、弾や敵の動きをスローにする

「エスプレイド」の後継作品。ボタン操作によって、処理落ちが発生したかのように弾や敵の動きをスローにすることができます。ちょうどこの本が出る（2004年6月）ころにPS2版が発売される予定です。

●ガンバード

・彩京：1994：AC/PS2/SS
・縦スクロール、溜め撃ち、近接攻撃

コミカルなキャラクターが魅力的なゲーム。途中からやや急に難しくなりますが、序盤はとっつきやすい難易度です。魔女っ娘好きの方には特にお勧め？

●ガンバード2

・彩京：1998：AC/DC/PS2
・縦スクロール、溜め撃ち、近接攻撃

ガンバードの続編。難易度が全体的に上がり、接近攻撃も加わりました。キャラクターの絵柄が変わったのは好みが分かれそうなところです。PS2版には本作と前作がいっしょに収録されています。

●ギガウィング

・カプコン/匠（たくみ）：1999：AC/DC
・縦スクロール（横画面）、弾を跳ね返す

弾を跳ね返す「リフレクトフォース」を駆使して戦うゲーム。弾幕をよけるというよりも、バリアをはりながら分厚い弾幕につっこんでいくプレイスタイルになります。どことなくパチンコで遊んでいるかのような感覚のあるゲームです。

●ギガウィング2

・カプコン/匠（たくみ）：2000：AC/DC
・縦スクロール（横画面）、弾を跳ね返す

「ギガウィング」の続編。「ボルカノン」と呼ばれるアイテム大量発生現象があり、前作よりも派手になっています。4人同時プレイにも対応しているDC版は、発動時に「リフレクトフォース」と叫びながら遊ぶのがお勧めです（発動タイミングをずらして、無敵時間を有効活用できるので）。

●ギャラガ

・ナムコ：1981：AC/PC/PS
・固定画面、編隊、敵に捕まってパワーアップ

敵の美しい編隊飛行が印象的なゲーム。大昔の作品でグラフィックもシンプルなのに、動きは現在でも美しく感じられます。捕らえられた自機を奪還するとショットが2連装になるのもユニークです。

●ギャラクシアン

・ナムコ：1979：AC/PS
・固定画面

ギャラガの前作にあたるゲーム。敵が整列しているようすはスペースインベーダーふうですが、ときおり列を離れて攻撃してくることが特徴です。ギャラガほど派手ではありませんが、敵は曲線を描いて飛行します。

●ギャプラス

・ナムコ：1984：AC/PS
・固定画面、編隊、敵を捕まえて味方にする

ギャラガの後継作品。敵を捕まえて味方にするなど、自機をパワーアップさせるルールがいろいろと追加されました。敵をお手玉のように撃つボーナスステージもユニークです。

●グラディウス

・コナミ：1985：AC/PC/PS/SS
・横スクロール、ミサイル、オプション、バリア

非常に有名な横スクロールゲームのシリーズ。レーザーやオプションといった豊富なパワーアップや、美しい旋律のBGMなどが魅力です。2004年夏には最新作の「グラディウスV」がPS2で発売される予定です。

●グラディウスII

・コナミ：1988：AC/PC/PS/SS
・強制横スクロール+任意縦スクロール、高速スクロール

グラディウスの続編。ステージが増え、自機やバリアの種類が選べるようになりました。シリーズのなかではグラディウスIIが最高傑作、と考えるプレイヤーも多いようです。

●グロブダー

・ナムコ：1984：AC/PC/PS
・固定画面、誘爆

「ゼビウス」に登場する「グロブダー」という戦車を主人公にした面クリアタイプのゲーム。スピーディーなゲーム展開と豊富な面数（99面）、面ごとに攻略パターンを見いだす楽しさなどが見どころです。

●極上パロディウス

・コナミ：1994：AC/PS/SS
・横スクロール、アイテムによる特殊攻撃

「パロディウス」の続編。キャラクターごとに異なる攻撃とパワーアップのパターンが用意されています。プレイの軽快さは前作そのままに、絵や演出がぐっと派手になりました。

●ケツイ 絆地獄たち

・ケイブ：2002：AC
・縦スクロール、ロックショット

「ロックショット」と呼ばれる追尾攻撃ができるゲーム。レーザーがロックショットに変わりましたが、内容としては「怒首領蜂」シリーズに近いといえます。

●サイヴァリア

・サクセス：2000：AC/PS2
・縦スクロール、弾幕、敵弾へのかすり

敵弾にかすって自機をレベルアップさせる「BUZZシステム」が特徴のゲーム。自機の当たり判定が極端に小さく、またレベルアップの瞬間は無敵になるので、針のような隙間でも通り抜けることができます。

●サイヴァリア リビジョン

・サクセス：2000：AC/PS2
・縦スクロール、弾幕、敵弾へのかすり

「サイヴァリア」のアレンジ版。1発の弾に何度でもかすれるようになるなど、基本部分のルールが変更されました。好きになると病みつきになるゲームです。最近は続編の「サイヴァリア2」がアーケードや各種コンシューマ機で発売されています。

●ザクソン

・セガ：1982：AC
・斜めスクロール、クォータービュー

右上から左下に向かってスクロールする独特の画面構成を持ったゲーム。自機を左右と上下（高さ方向）に操作して、高さのある障害物で構成されたステージを進みます。

●式神の城

・アルファシステム：2001：AC/PC/PS2/XB
・縦スクロール、敵弾へのかすり、ワープする自機

テンポのよいゲーム展開や個性的なキャラクター、豊富な演出などが好評を博したゲーム。高密度の弾幕やかすりによるパワーアップなど、最近のシューティングらしい要素をひとそろい楽しむことができます。

●式神の城Ⅱ

・アルファシステム：2003：AC/DC/GC/PS2/XB
・縦スクロール、敵弾へのかすり

「式神の城」の続編。NAOMI基板に変わったことで、グラフィックが細かく美しくなりました。現行コンシューマ機のほとんどに移植されているのもよい点です。

●疾風魔法大作戦

・ライジング/エイティング：1994：AC/SS
・縦スクロール、スクロール速度の調節

「魔法大作戦」の続編。レース要素を取り入れていて、ただ進むだけではなく、ライバル機との競争に勝つ必要もあります。自機の上下位置によってスクロール速度が変化します。

●Gダライアス

・タイトー：1997：AC/PC/PS
・横スクロール、レーザー同士の衝突、敵を捕まえて味方にする

横スクロールゲームとして有名な「ダライアス」シリーズの1つ。3Dグラフィックを使用しています。シリーズの伝統である面分岐や、海洋生物を模した巨大なボスなどは健在です。

●スカイキッド

・ナムコ：1985：AC/PS
・左から右への横スクロール

珍しい左→右スクロールのゲーム。古きよきナムコのゲームらしく、小さなキャラクターなのにきめ細かく、かつコミカルに動くのが魅力です。レバーの入力方向で宙返りのパターンが違うなど、凝って作ってあります。

●スペースインベーダー

・タイトー：1978：AC/PC/PS/PS2/SS
・固定画面

社会現象にもなったアーケードゲーム黎明期の大ヒット作。整列した状態で迫ってくるインベーダーを撃ち落とします。インベーダーの動きに比べて自機の速度が遅く、ショットも単発なので、今遊んでみるとかなりシビアなゲームです。

●スペースハリアー

・セガ：1985：AC/DC/PS2/SS
・疑似3D

セガの体感ゲームシリーズの1つ。人間タイプの自機を動かして、疑似3D画面のフィールドを進んでいきます。当時のゲームとしては破格に美しいグラフィックで人気を集めました。

●スペースボンバー

・彩京：1998：AC
・固定画面、敵をつかまえて投げる、敵をつかまえて味方にする

疑似3Dで描かれた宇宙人や動物などを相手に、敵をつかまえて投げたり、味方にしたりといった方法で戦うコミカルなゲーム。うかうかしていたらゲームセンターから消えてしまったのが残念です。面の合間に入る寸劇も笑えます。

●セクシーパロディウス

・コナミ：1996：AC/PS
・横スクロール、やられ原因の表示

「パロディウス」シリーズの1つ。各面に「指定されたアイテムを拾え」や「特定の敵を多く倒せ」といったノルマが設定されているのがユニークです。ゲームの難易度やテンポも非常によく調整されています。

●ゼビウス

・ナムコ：1982：AC/PC/PS
・縦スクロール、照準による爆撃、弱点を持つボス

縦スクロールシューティングゲームとしておそらくもっとも有名な作品。美しいグラフィック、スクロールする広大なマップ、個性的な敵の数々など、それまでのゲームにない要素をいくつも持っていたゲームです。

●ゼビウス3D/G

・ナムコ：1995：AC/PS
・奥行きスクロール

「ゼビウス」の続編の1つ。斜め見下ろし視点の3Dグラフィックを使用しています。2人同時プレイも可能です。なお、コクピット視点の3Dシューティングになったゼビウスとしては「ソルバルウ」があります（アーケードのみ）。

●ゼロガンナー

・彩京：1997：AC
・奥行きスクロール、方向切り替えによる全方位射撃

セガのMODEL2基板を使用した斜め見下ろし型視点の3Dシューティングゲーム。自機の方向を切り替えることで、全方位に攻撃することができます。

●ゼロガンナー2

・彩京：2001：AC/DC
・奥行きスクロール、方向切り替えによる全方位射撃

「ゼロガンナー」の続編。全方位攻撃のルールは前作を踏襲しています。「ゼロガンナー」シリーズは、横画面にもかかわらず縦方向の狭さを感じさせないという点でも優れています。

●ソニックウィングス2

・ビデオシステム：1994：AC/PS
・縦スクロール（横画面）

NEOGEO（MVS基板）用に開発された横画面、縦スクロールのゲーム。前作の「ソニックウィングス」は縦画面で、続編の「ソニックウィングス3」は横画面です。PS版には3作品とともに収録されています。

●タイムパイロット

・コナミ：1982：AC/PS
・全方向スクロール

自機が画面中央に表示されていて、画面が全方向にスクロールするタイプのゲーム。自機のなめらかな動きと多重スクロールがあいまって、独特の飛行感覚が味わえます。

●ダライアス外伝

・タイトー：1994：AC/PC/PS/SS
・横スクロール、敵を捕まえて味方にする

「ダライアス」シリーズの3作目。初代では3画面構成だったのが1画面構成になりましたが、一般的な画面構成になったのを歓迎するプレイヤーも多いようです。難易度は高めで、特に連射能力（あるいは連射装置）が必須となっています。

●チェルノブ

・データイースト：1988：AC
・横スクロール、地上を歩く自機

地上を走ったりジャンプしたりする人間が自機のゲーム。ジャンプが2種類あったり、ショットの方向が変えられたりと、操作性は独特です。「戦う人間発電所」というやや危険な副題がついています。

●超時空要塞マクロス

・バンプレスト：1992：AC
・縦スクロール、変形する自機

同名の人気アニメをモチーフにしたゲームですが、ゲーム単体として見てもよくできています。原作どおりに自機を3形態に変形させながら戦います。

●テラクレスタ

- 日本物産：1985：AC/PC
- 縦スクロール、合体する自機

「ムーンクレスタ」の後継作品。ゲーム内容はかなり変わりましたが、自機の合体があるのは前作ゆずりです。マップ上にある自機のパーツを拾って、5機まで合体することができます。

●ツインビーヤッホー

- コナミ：1995：AC/PS/SS
- 縦スクロール（横画面）、近接攻撃、味方をつかんで投げる、味方に接近してショットを強化

「ツインビー」シリーズの1作品。ほかの「ツインビー」と同じく縦スクロールですが、この作品は横画面を使用しています。2人プレイではいろいろな協力攻撃が楽しめます。

●出たな!! ツインビー

- コナミ：1991：AC/PS/SS
- 縦スクロール、味方に接近してショットを強化

縦スクロールゲームとして有名な「ツインビー」シリーズの1作品。パステル調の美しいグラフィックと軽快な音楽が印象的なゲームです。

●首領蜂（どんぱち）

- アトラス/ケイブ：1995：AC/PS/SS
- 縦スクロール、レーザー、コンボボーナス

「怒首領蜂」「怒首領蜂 大往生」などの「蜂」シリーズの原点となるゲーム。後継作品に比べると地味ですが、「ボタン連打でショット、押しっぱなしでレーザー」というルールはこのゲームで確立されました。

●怒首領蜂（どどんぱち）

- アトラス/ケイブ：1997：AC/PS/SS
- 縦スクロール、レーザー、コンボボーナス

「首領蜂」の続編。ショットとレーザーの撃ち分けはそのままに、ゲーム全体をグレードアップさせた作品です。弾幕が厚く難易度も高めですが、ゲームのテンポがよく、当たり判定などの調整も絶妙なので、不思議なほど気持ちよく遊ぶことができます。

●怒首領蜂 大往生（どどんぱち だいおうじょう）

- ケイブ：2002：AC/PS2
- 縦スクロール、レーザー、コンボボーナス

「怒首領蜂」の続編。基本的な内容は「怒首領蜂」と同じながらも、弾が速く弾幕も厚いため、難易度は大幅にアップしている感があります。PS2版は豊富なプレイ条件設定機能やリプレイ機能を加え、攻略DVDも同梱した好移植です。

●ドラゴンセイバー

・ナムコ：1990：AC/PS
・縦スクロール、溜め撃ち

「ドラゴンスピリット」の続編。自機はドラゴンで、アイテムを取ることで首の数が3本まで増えて火力がアップします（当たり判定も大きくなります）。音楽のよさも魅力です。

●バトルガレッガ

・ライジング/エイティング：1996：AC/SS
・縦スクロール、自爆による難易度調整

難易度調整に特徴があるゲーム。ゲームの進行とともに自動的に上がっていく難易度を下げるために、「残機を稼いだうえで自爆する」という独特のプレイスタイルが要求されます。

●パロディウスだ！

・コナミ：1990：AC/PS/SS
・横スクロール、アイテムによる特殊攻撃

「パロディウス」シリーズの1作目。ゲームのいたるところに「グラディウス」シリーズのパロディを見ることができます。コミカルなキャラクター、適度な難易度、軽快なゲーム展開なども魅力です。

●パンツァードラグーン

・セガ：1995：PC/SS/XB
・3D、ロックオンレーザー

セガサターンの初期に発売された3Dシューティングゲーム。奥行き方向に強制スクロールするステージを巨大なドラゴンを操って攻略します。美しい画面と音楽が特徴です。なお、Xboxの「パンツァードラグーン オルタ」には本作も収録されています。

●B-ウィング

・データイースト：1984：AC
・縦スクロール、合体する自機

8種類の「ウィング」を自機に装着してパワーアップするゲーム。装着したウィングによって、発射できる弾の方向や種類が変わります。

●ビューポイント

・サミー：1992：AC
・斜めスクロール、クォータービュー

「ザクソン」と同じく右上から左下にスクロールする画面構成のゲーム。自機が上下左右に移動する「ザクソン」とは違い、自機は前後左右に移動します。クォータービューですが、プレイの感覚は縦スクロールゲームに近いものがあります。

●プーヤン

・コナミ：1982：AC/PS
・固定画面

ブタの親子がオオカミを退治するというコミカルなゲームです。ゴンドラに乗ったブタを上下に操作して、風船に乗って襲ってくるオオカミを矢で撃ち落とします。矢の連射が効かないうえに、風船の部分を狙って撃たないといけないので、ショットを無尽蔵に撃てる最近のゲームとはかなり違った味わいがあります。

●フェリオス

・ナムコ：1998：AC
・縦スクロール、溜め撃ち、壁で反射するショット、回転

ギリシャ神話をモチーフにしたゲーム。アポロンを操作して、捕らわれたアルテミスを救出しに行くという設定です。ステージの合間ではアルテミスが「早く助けにきて」と切々と訴えます（プレイヤーへの餌？）。拡大縮小や回転を使った演出もあります。

●フォーメーションZ

・ジャレコ：1984：AC/PC/PS
・横スクロール、変形する自機

自機をロボット形態と戦闘機形態に変形させることができるゲーム。なめらかな変形モーションが魅力的です。戦闘機形態では時間とともにエネルギーを消費してしまうなど、シビアな面もあります。

●プロギアの嵐

・カプコン/ケイブ：2001：AC
・横スクロール、ロックショット

「怒首領蜂」シリーズのケイブが開発した横スクロールゲーム。ケイブらしい分厚い弾幕と、それでもなぜかよけられてしまう当たり判定の絶妙さが魅力です。同乗する女の子の好感度を稼ぐという、恋愛シミュレーションふうの要素もあります。

●ボーダーダウン

・グレフ：2003：AC/DC
・横スクロール、レーザー同士の衝突

元タイトーのスタッフが立ち上げたグレフの横スクロールゲーム。レーザー干渉（レーザー同士の衝突）による攻撃は「Gダライアス」ゆずりです。ミスするたびに難易度が違う「ボーダー」に移行する、という独特なルールもあります。

●ボスコニアン

・ナムコ：1981：AC/PC/PS
・全方向スクロール

全方向にスクロールする宇宙空間で敵と戦うゲーム。自機は画面の中央に固定されています。全方向スクロールのゲームらしく、敵に近づいたり離れたりしながら戦います。

●ミサイルコマンド

・アタリ：1980：AC/PC
・固定画面、誘爆

トラックボールで照準を動かして、上空から降り注ぐミサイルを撃ち落とすゲーム。ミサイルを破壊すると爆発し、その爆発でほかのミサイルを誘爆させることができます。誘爆を使って上手に弾幕をはることがポイントです。

●ムーンクレスタ

・日本物産：1980：AC/PC/PS
・固定画面、合体する自機

自機の合体が特徴の固定画面シューティングゲーム。性能が異なる3機の機体が登場します。特定の面をクリアすると合体ステージがあり、合体に成功すると各機体の攻撃能力がミックスされます。

●メタルスラッグ

・SNK/ナスカ：1996：AC/PS/SS
・横スクロール、地上を歩く自機

人間が主人公のアクション&シューティングゲーム。グラフィックの細かさやアニメーションの緻密さ、キャラクターのアクションの豊富さなどが人気を集め、多くの続編が作られています。

●レイクライシス

・タイトー：1998：AC/PC/PS
・奥行きスクロール、ロックオンレーザー

「レイストーム」の続編。基本的なルールはそのままに、「侵食率」によって難易度が変化するという新しい要素も取り入れています。PS用には前作と今作を同時収録した移植版が出ています（ただし、単体移植版にあった一部のモードが削られています）。

●レイストーム

・タイトー：1996：AC/PC/PS/SS
・奥行きスクロール、ロックオンレーザー

「レイフォース」の続編。縦スクロールだった前作とは違い、3Dグラフィックを使った横画面の奥行きスクロールゲームになりました。美しい画面と派手な演出が魅力的なのですが、弾が見づらい瞬間もあります。

●レイフォース

・タイトー：1993：AC/PC/SS
・縦スクロール、ロックオンレーザー

ロックした敵を誘導レーザーで破壊する「ロックオンレーザー」を取り入れたゲーム。絵、音楽、演出、ゲーム性のいずれの面でも完成度が高い作品です。2Dグラフィックなのにもかかわらず、レーザーの動きが非常になめらかなのにも驚きます。

Appendix 3 索引

■英数字

DDA 15,26
n-way弾 33
n-way弾の発射 34

■あ行

アームの動き 182
アイテム 169
あそび 82
当たり判定 3,31,50,234
当たり判定処理 31,98,108,138,149,198,201,234
アナログスティック 82
アナログ入力 82
アルファ合成 220
安全地帯 67
移動可能範囲 76
色を変える 195
動きをスローにする 205
渦巻き弾 62
撃ち返し 257
永久パターン 297
円形弾 36
奥行きスクロール 268
オプション 104

■か行

回転 281
回転弾 58,61
回転半径を変化させる 59
開発環境 5
かすり判定 203
加速弾 54
合体 91
画面の配色 306
画面構成 262
奇数パターン 33,36
軌道データ 226,228
軌道を描いて飛ぶ敵 226
強制スクロール 265
強制縦スクロール 276
強制横スクロール 279
近接攻撃 165
偶数パターン 33,36
空中に現れる敵 220
ゲームバランス 164
ゲームモード 308
桁数 290
限定横スクロール 276
攻撃の予兆 66
高速スクロール 283
固定画面 265
固定小数点数 18,20
固定砲台 224
コマンドショット 131
コマンドの入力時間 134
コマンド入力の判定 133
コマンド入力のゆらぎ 132
コンティニュー 298

■さ行

残機ボーナス 297
残ボムボーナス 297
自機 2
自機に力をかける 284
自機の移動 74,82
自機の色を切り替えて弾を吸収する 195
自機を取り返してパワーアップ 188
シフト演算 20
シューティングゲームの基本構成 2
シューティングゲームの作成手順 4
シューティングゲームの進行 3

自由に動かせる照準 207
照準 141,144,207,210
照準の移動 144
照準を使った爆撃 141
照準を点滅させる 142
触手 246
触手の移動範囲 246
ショット 114,151
ショットの位置の判定 137
ショットの威力 140
ショットの移動 137
水中の移動 98
スクロール 262,265
スクロール速度の計算 287
スクロール速度の調整 286
スコア 3,290,297
スコアの再現性 297
スコアランキング 299
スピード調整 88
セミオート連射 119
旋回 157
旋回角度の計算 158
旋回方向の選択 41
全方向スクロール 269
全方位射撃 157
その場復活 165

■た行

ターンマーカー 157
多関節 250
多重スクロール 272
縦スクロール 266
縦画面 262
多倍長数と整数の乗算 293
多倍長数に整数を加算 292
多倍長数に多倍長数を加算 292
多倍長数の初期化 291
弾 2
弾の位置の判定 29
弾の移動 15,18,21
弾の消去 29,61
弾の速度 11,13
弾の動き 10,70
弾を吸収する 195
溜め撃ち 117,119,124
溜めパワー 117,126
地形で反射するショット 151
地形で反射するレーザー 153
地形に沿って飛ぶミサイル 148
地形による反射 152
地上を歩く自機 93
チップ 270
チップを使った背景の表示 271
直進するビーム 65
テーブルを使った方向弾 24
停止する誘導弾 64
敵 2
敵弾に自機をかすらせる 202
敵弾をショットとして跳ね返す 197
敵の堅さ 223
敵の出現 218
敵の消失 218
敵の耐久力 214
敵の速さ 223
敵の編隊 229
敵をつかまえて投げる 181
敵をつかまえて味方にする 185
特殊攻撃 169

■な行

投げつけられた敵の動き 184
投げられた機体の移動 179
斜め移動 80
斜めスクロール 267
難易度 303
逃げる誘導弾 64
任意スクロール 265
任意横スクロール 279
狙い撃ち弾 10,61
狙い撃ち弾+回転弾 61

■は行
バーサーク状態 174
背景 2,270
背景の表示 270
破壊 214,216
爆撃 141
跳ね返し 197
バリア 108,110
パワーアップ 101,188,202
武器 2
武器の攻撃力 214
武器の切り替え 155
複雑な形の当たり判定 234
複数人数プレイ 308
プラットフォーム 5
フレームレートの調整 309
プログラムが動く仕組み 11
分裂弾 38
変形 96
編隊の移動 229
編隊の出現 232
方向切り替え 157
方向弾 23,26
星のスクロール 274
ボスキャラ 237,239,241,244
ボスキャラの構造 267
ボスキャラの行動 239
ボスキャラの合体 241
ボスキャラの分離 241
ボスキャラの変形 244
ボム 162,164

■ま行
味方がいる方向に強いショットを撃つ 192
味方に当てたショットを強化する 193
味方に接近したかどうかの判定 190
味方に接近してショットを強化する 190
味方になった敵の動き 185
味方をつかんで投げる 178
ミサイル 52,148
ミサイルが命中する条件 150
見やすい重ね合わせ 307
無敵状態 172
戻り復活 165,298

■や行
やられた原因の表示 307
やられ判定 203
誘導弾 39,63
誘導レーザー 46
誘導レーザーの描画 49
誘爆 167
横スクロール 267
横画面 262

■ら行
落下弾 56
乱数 302
乱数の種 302
リプレイ 299
レーザー 126,153,200
レーザー同士をぶつける 200
レーザーの動き 145
レーザーの描画 154
連射 116,119,126
連射ゲージ 120,126
連射の停止 121
ロール表示 77,79
ロック 144
ロックオンレーザー 144
ロックショット 129

■わ行
ワープ移動 85

■リスト（掲載順）
狙い撃ち弾の初期化 14
狙い撃ち弾の移動 14
DDAを使った狙い撃ち弾の初期化 16
DDAを使った狙い撃ち弾の移動 17
固定小数点数を使った狙い撃ち弾の初期化 22
固定小数点数を使った狙い撃ち弾の移動 22

方向弾の初期化……23
方向弾の移動……24
16方向に飛ぶ速さ3の弾の発射……25
DDAを使った方向弾の初期化……26
位置のテーブルを作る……28
位置のテーブルを使った方向弾の初期化……28
速度ベクトルを回転させる……34
n-way弾の初期化……35
円形弾の初期化……37
誘導弾の移動……39
旋回角度を制限した誘導弾の移動……44
誘導レーザーの移動……47
誘導レーザーの発射……48
ミサイルの発射……53
速さと速度の更新……56
落下弾の移動……57
回転弾の移動……59
半径が変化する回転弾の移動……60
自機の移動……75
移動可能範囲を考慮した自機の移動……77
ロール表示を行うプログラム……78
3Dグラフィックを使って
ロール表示を行うプログラム……80
上下左右と斜めの速さを同じにした移動……81
アナログ入力で自機を移動させるプログラム 84
ワープ移動……86
ボタンによるスピード調整……88
アイテムによるスピード調整……90
合体する味方の動き……92
地上を歩く自機の動き……94
変形する自機……97
水中の移動(自機の高度を調べる場合)……100
水中の移動(水との当たり判定処理を行う場合)
……100
ゲージを使ったパワーアップ……103
オプションの初期化と移動……107
バリア……109
ボタンで張るバリア……111
基本のショット操作……116
連射……117
溜め撃ち……118
連射と溜め撃ちの共存……122
ボタンを離して溜める溜め撃ち……125
連射とレーザーの共存……128
ロックショット……130
コマンドショット……135
ショットの移動……138
ショットの当たり判定処理……139
敵との距離によるショットの威力の違い……140
照準を使った爆撃……143
ロックオンレーザーの発射……146
地形に沿って飛ぶミサイル……150
地形で反射するショット……152
地形で反射するレーザー……154
武器の切り替え……156
方向切り替えによる全方位射撃……159
ボム……163
近接攻撃……166
誘爆……168
アイテムによる特殊攻撃……170
無敵状態……173
バーサーク状態……176
投げられた機体の移動……180
アームの動き……182
投げつけられた敵の動き……184
味方になった敵の動き……186
自機をつかまえる敵の動き……189
味方と接近してショットを強化する……191
味方がいる方向に強いショットを撃つ……193
味方に当てたショットを強化する……194
自機の色を切り替えて弾を吸収する……196
敵弾をショットとして跳ね返す……199
レーザー同士をぶつける……201
敵弾に自機をかすらせてパワーアップする
……204
敵や弾の動きをスローにする……206
自由に動かせる照準……208
自機と照準を同時に動かす……210
破壊できる敵の処理……215
破壊できないものも含めた敵の処理……217
敵の出現と消去……219
空中に現れる敵……221

固定砲台 225
用意した軌道データを使って敵を動かす 227
軌道データとプログラムを
組み合わせて敵を動かす 228
編隊の移動 231
編隊の出現 233
複雑な形の当たり判定 236
ボスキャラの構造 238
ボスキャラの行動 240
ボスキャラの分離と合体 242
ボスキャラの変形 245
触手 249
多関節 255
撃ち返し 259
チップを使った背景の表示 271
多重スクロール 274
星のスクロール 275
強制縦スクロール+限定横スクロール 278
上下左右がつながった背景の表示 280
回転 283
スクロールと自機に対する力 285
スクロール速度の計算 287
多倍長演算 295
乱数 301
乱数の生成 303

■デモプログラム

狙い撃ち弾 10,312
狙い撃ち弾2 10,312
狙い撃ち弾（DDA） 15,312
狙い撃ち弾（固定小数点数） 18,312
方向弾 23,312
テーブルを使った方向弾 24,312
方向弾（DDA） 26,312
n-way弾 33,312
円形弾 36,313
分裂弾 38,313
簡易誘導弾 39,313
誘導弾 39,313
誘導レーザー 46,313
誘導レーザー2 46,313
誘導ミサイル 52,313
加速n-way弾 54,313
加速誘導レーザー 54,313
落下弾 56,313
落下ミサイル 56,314
回転弾 58,314
回転弾2 58,314
狙い撃ち弾＋回転弾 61,314
渦巻き弾 62,314
停止する誘導弾 63,314
逃げる誘導レーザー 63,314
直進するビーム 65,314
安全地帯 67,314
安全地帯2 67,314
乱数 70,315
乱数2 70,315
ワープ移動 85,315
ボタンによるスピード調整 88,315
アイテムによるスピード調整 89,315
合体する自機 91,315
地上を歩く自機 93,315
変形する自機 96,315
水中の移動 98,316
オプション 104,316
バリア 108,316
基本のショット操作 114,316
連射 116,316
溜め撃ち 117,316
セミオート連射 119,316
ボタンを離して溜める溜め撃ち 124,317
連射とレーザーの共存 126,317
ロックショット 129,317
コマンドショット 131,317
照準を使った爆撃 141,317
ロックオンレーザー 144,317
地形に沿って飛ぶミサイル 148,317
地形で反射するショット 151,317
地形で反射するレーザー 153,317
方向切り替えによる全方位射撃 157,317
ボム 162,318
パンチ 165,318

誘爆 167,318
アイテムによる特殊攻撃 169,318
無敵状態 172,318
バーサーク 174,318
味方をつかんで投げる 178,318
敵をつかんで投げる 181,318
敵をつかまえて味方にする 185,319
敵につかまった自機を
取り返してパワーアップする 188,319
味方に接近してショットを強化する 190,319
味方がいる方向に強いショットを撃つ 192,319
味方に当てたショットを強化する 193,319
自機の色を切り替えて弾を吸収する 195,319
敵の弾をショットとして跳ね返す 197,319
レーザー同士をぶつける 200,319
敵弾に自機をかすらせる 202,319
弾や敵の動きをスローにする 205,319
自由に動かせる照準 207,320
自機と照準を同時に動かす 210,320
破壊できる敵 214,320
破壊できない敵 216,320
空中に現れる敵 220,320
固定砲台 224,320
軌道を描いて飛ぶ敵 226,320
敵の編隊 229,320
ボス 237,321
触手 246,321
多関節 250,321
撃ち返し 257,321
背景の表示 270,321
多重スクロール 272,321
星のスクロール 274,321
強制縦スクロール＋限定横スクロール 276,321
強制横スクロール＋任意縦スクロール 279,322
回転 281,322
高速スクロール 283,322
プレイヤーによるスクロール速度の調整
......286,322

■引用ゲーム

R-TYPE 153,267,323
R-TYPE FINAL 308,323
アサルト 281,323
アフターバーナー 269,324
斑鳩 195,298,324
SDI 210,324
エスプレイド 110,324
エスプガルーダ 110,205,324
ガンバード 165,325
ガンバード2 165,325
ギガウィング 197,265,290,325
ギガウィング2 197,265,290,325
ギャラガ 188,226,229,265,325
ギャラクシアン 265,326
ギャプラス 226,229,326
グラディウス 101,104,148,267,326
グラディウスII 279,326
グロブダー 110,167,326
ケツイ 絆地獄たち 129,327
極上パロディウス 174,326
サイヴァリア 32,131,202,327
サイヴァリア リビジョン 203,327
ザクソン 267,327
式神の城 32,85,202,207,327
式神の城II 32,202,207,327
疾風魔法大作戦 286,328
Gダライアス 185,200,328
スカイキッド 267,328
スペースインベーダー 265,328
スペースハリアー 269,328
スペースボンバー 181,185,328
セクシーパロディウス 307,329
ゼビウス 141,144,216,220,237,266,329
ゼビウス3D/G 268,329
ゼロガンナー 157,329
ゼロガンナー2 157,329
ソニックウィングス2 265,330
タイムパイロット 268,330
ダライアス外伝 185,330
チェルノブ 93,330
超時空要塞マクロス 96,330
テラクレスタ 91,331

ツインビーヤッホー ...165,178,190,192,193,331
出たな!!ツインビー 190,331
首領蜂 126,331
怒首領蜂 126,298,331
怒首領蜂 大往生 126,331
ドラゴンセイバー 124,331
バトルガレッガ 304,332
パロディウスだ！ 101,169,332
パンツァードラグーン 144,332
B-ウィング 91,332
ビューポイント 267,332
フェリオス 117,151,281,333
フォーメーションZ 96,333
プーヤン 265,333
プロギアの嵐 129,333
ボーダーダウン 200,333
ボスコニアン 268,334
ミサイルコマンド 167,334
ムーンクレスタ 91,334
メタルスラッグ 93,334
レイクライシス 144,268,334
レイストーム 55,144,268,335
レイフォース 55,144,335

おわりに

「シューティングゲームを遊ぶ人は、一度くらい『自分でシューティングゲームを作ろう』と思ったことがあるはず！」という考えのもとに、普通のゲームプログラミング入門書とは違った切り口で進めてきた本書でしたが、お楽しみいただけたでしょうか。本書がシューティングゲームに関してあれこれ考えたり、楽しく遊んだり、熱く語ったり、プログラムを書いたり、一層シューティングゲームが好きになったりするための手助けになれば幸いです。

末筆になりましたが、本書の執筆にあたり多くのアドバイスやヒントをくださった森田健郎氏に御礼申し上げます。また、校正を手助けしてくれた妻に感謝します。

松浦 健一郎

■著者プロフィール

松浦 健一郎(まつうら けんいちろう)

東京大学工学系研究科電子工学専攻修士課程を修了後、研究所勤務を経て、現在は趣味と実益を兼ねつつフリーのプログラマ＆ライターとして活動中。著書に『はじめてのJBuilder4』『Delphi DB&Webプログラミング』『はじめてのJBuilder6』『デスクトップマスコットを作ろう!!』、雑誌連載に『C MAGAZINE』誌の「ゲーム・ノ・シクミ」(連載中)『プログラムのレシピ』『Java Programming Tips』などがある（いずれもソフトバンク刊)。関心と仕事の範囲はプログラミングを中心にコンピュータ全般に及ぶが、最も興味がある分野はプログラミング言語作りとゲーム作り。

著者Webサイト「ひぐぺん工房」
http://cgi32.plala.or.jp/higpen/gate.shtml

本書をお読みになったご感想、ご意見を次のURLからお寄せください。

http://isbn.sbpnet.jp/27316

なお、掲示期間の超過したアンケートは表示されない場合がありますので、あらかじめご了承ください。
本書に関するサポート情報やお問い合わせ受け付けフォームも掲載予定ですので、あわせてご利用ください。

シューティングゲーム アルゴリズム マニアックス

2004年6月18日　初版第1刷発行
2005年4月22日　初版第5刷発行

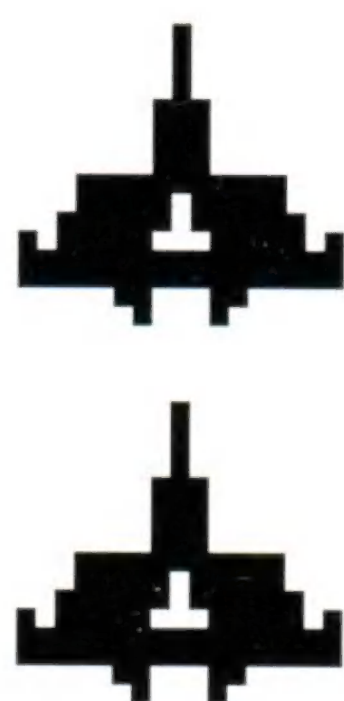
著　者・・・・・・・・・・松浦 健一郎（まつうら けんいちろう）

発行者・・・・・・・・・稲葉 俊夫
発行所・・・・・・・・・ソフトバンクパブリッシング株式会社
〒107-0052　東京都港区赤坂4-13-13
電話　販売局　03 (5549) 1200
　　　編集局　03 (5549) 1143

カバー／本文デザイン・・クニメディア株式会社
組　版／印　刷・・・・・クニメディア株式会社

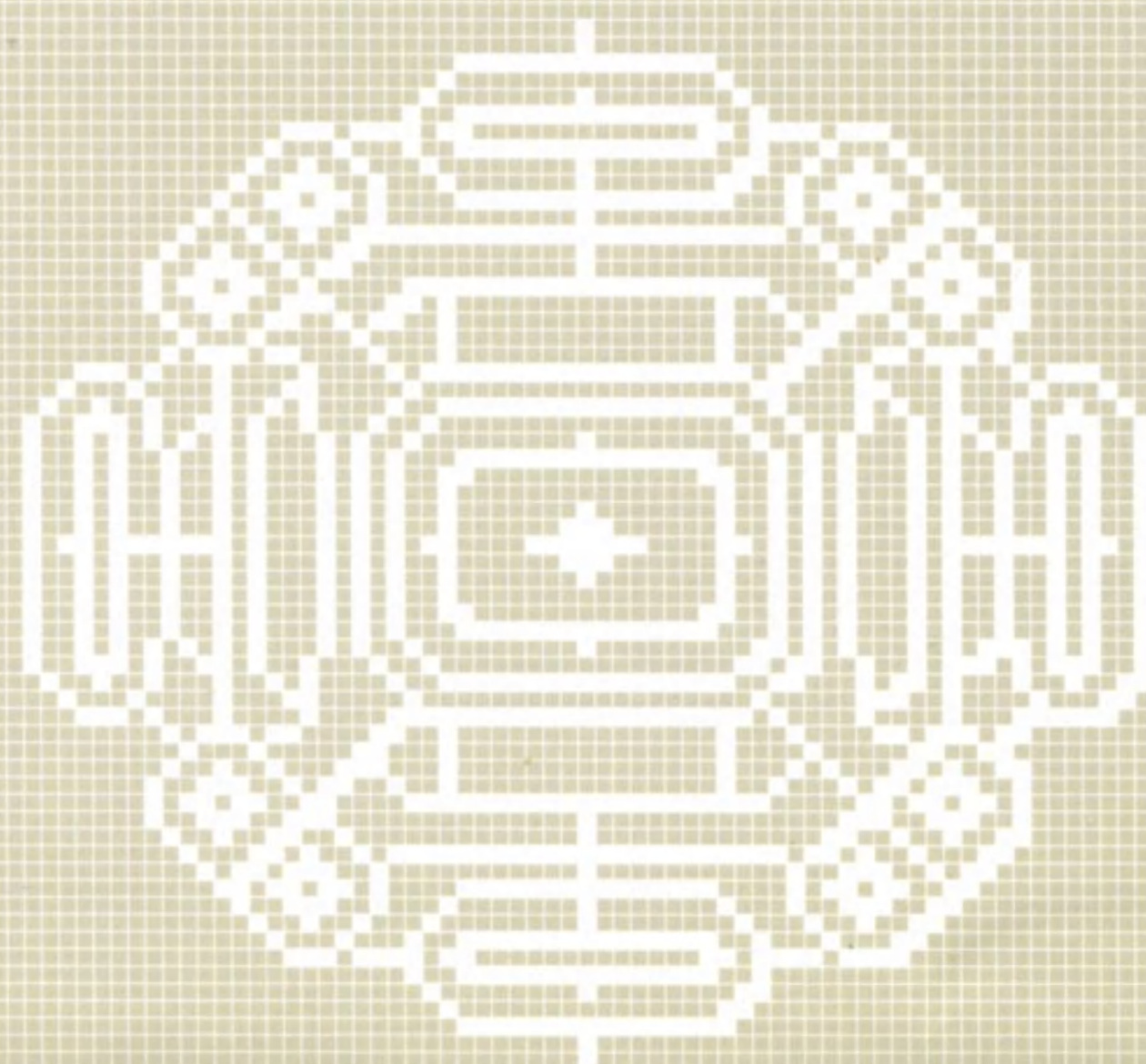
落丁本、乱丁本は小社販売局にてお取り替えいたします。
定価はカバーに記載されております。
本書内容に関するご質問などは、ご面倒でも小社C MAGAZINE編集部まで、
必ず書面にてご連絡くださいますようお願いいたします。

Printed In Japan　　ISBN4-7973-2731-6

SOFT BANK
Publishing
ソフトバンクパブリッシング

シューティングゲーム アルゴリズム マニアックス
ShootingGame Algorithm Maniax
松浦健一郎 著

SOFT BANK
Publishing

SOFT BANK Publishing
ソフトバンクパブリッシング

ISBN4-7973-2731-6
C0055 ¥2800E
9784797327311
定価 本体2,800円 +税
1920055028004

シューティングゲーム アルゴリズム マニアックス
ShootingGame Algorithm Maniax

松浦健一郎 著

SOFT BANK Publishing

シューティングゲーム アルゴリズム マニアックス

COMPACT disc

SOFT BANK Publishing

ShootingGame Algorithm Maniax